dask-ml Documentation

Release 0.1

Dask developers

Jul 19, 2018

Get Started

Dask-ML provides scalable machine learning in Python using Dask alongside popular machine learning libraries like Scikit-Learn.

You can try Dask-ML on a small cloud instance by clicking the following button:

```
import dask.dataframe as dd
df = dd.read_parquet('...')
data = df[['age', 'income', 'married']]
labels = df['outcome']

from dask_ml.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(data, labels)
```

CHAPTER 1

What does this offer?

See the navigation pane to the left for a list of categories of functionality.

CHAPTER 2

How does this work?

Modern machine learning algorithms employ a wide variety of techniques. Scaling these requires a similarly wide variety of different approaches. Generally solutions fall into the following three categories:

2.1 Parallelize Scikit-Learn Directly

Scikit-Learn already provides parallel computing on a single machine with Joblib. Dask can now step in and take over this parallelism for many Scikit-Learn estimators. This works well for modest data sizes but large computations, such as random forests, hyper-parameter optimization, and more.

```
from dask.distributed import Client
client = Client()  # start a local Dask client

import dask_ml.joblib
from sklearn.externals.joblib import parallel_backend
with parallel_backend('dask'):
    # Your normal scikit-learn code here
```

See *Dask-ML Joblib documentation* for more information.

Note that this is an active collaboration with the Scikit-Learn development team. This functionality is progressing quickly but is in a state of rapid change.

2.2 Reimplement Scalable Algorithms with Dask Array

Some machine learning algorithms are easy to write down as Numpy algorithms. In these cases we can replace Numpy arrays with Dask arrays to achieve scalable algorithms easily. This is employed for *linear models*, *pre-processing*, and *clustering*.

```
from dask_ml.preprocessing import Categorizer, DummyEncoder
from dask_ml.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(data, labels)
```

2.3 Partner with other distributed libraries

Other machine learning libraries like XGBoost and TensorFlow already have distributed solutions that work quite well. Dask-ML makes no attempt to re-implement these systems. Instead, Dask-ML makes it easy to use normal Dask workflows to prepare and set up data, then it deploys XGBoost or Tensorflow *alongside* Dask, and hands the data over.

```
from dask_ml.xgboost import XGBRegressor

est = XGBRegressor(...)
est.fit(train, train_labels)
```

See *Dask-ML + XGBoost* or *Dask-ML + TensorFlow* documentation for more information.

CHAPTER 3

Scikit-Learn API

In all cases Dask-ML endeavors to provide a single unified interface around the familiar NumPy, Pandas, and Scikit-Learn APIs. Users familiar with Scikit-Learn should feel at home with Dask-ML.

3.1 Installation

3.1.1 Conda

`dask-ml` is available on conda-forge and can be installed with

```
conda install -c conda-forge dask-ml
```

3.1.2 PyPI

Wheels and a source distribution are available on PyPI and can be installed with

```
pip install dask-ml
```

Optional Dependencies

Certain modules have additional dependencies:

module	Additional Dependencies
`dask_ml.xgboost`	xgboost, dask-xgboost
`dask_ml.tensorflow`	tensorflow, dask-tensorflow

These additional dependencies will need to be installed separately. With pip, they can be installed with

```
pip install dask-ml[xgboost]    # also install xgboost and dask-xgboost
pip install dask-ml[tensorflow]
pip install dask-ml[complete]   # install all optional dependencies
```

3.2 Examples

This is a set of runnable examples demonstrating how to *use* Dask-ML.

3.2.1 Hyper-parameter Search

Most scikit-learn estimators have a set of *hyper-parameters*. These are parameters that are not learned during estimation; they must be set ahead of time.

The `dask-searchcv <http://dask-searchcv.readthedocs.io/en/latest/>`__ is able to parallelize scikit-learn's hyper-parameter search classes cleverly. It's able to schedule computation using any of dask's schedulers.

```
In [2]: import numpy as np

        from time import time
        from scipy.stats import randint as sp_randint
        from scipy import stats

        from distributed import Client
        import distributed.joblib

        from sklearn.externals import joblib
        from sklearn.datasets import load_digits
        from sklearn.linear_model import LogisticRegression

        from dask_searchcv import GridSearchCV, RandomizedSearchCV
        from sklearn import model_selection as ms
        import matplotlib.pyplot as plt

        client = Client()
```

This example is based off this scikit-learn example.

```
In [3]: # get some data
        digits = load_digits()
        X, y = digits.data, digits.target
```

We'll fit a `LogisticRegression`, and compare the `GridSearchCV` and `RandomizedSearchCV` implementations from `scikit-learn` and `dask-searchcv`.

Grid Search

Grid-search is the brute-force method of hyper-parameter optimization. It fits each combination of parameters, which can be time consuming if you have many hyper-parameters or if you have a fine grid.

To use grid search from scikit-learn, you create a dictionary mapping parameter names to lists of values to try. That `param_grid` is passed to `GridSearchCV` along with a classifier (`LogisticRegression` in this example). Notice that `dask_searchcv.GridSearchCV` is a drop-in replacement for `sklearn.model_selection.GridSearchCV`.

```
In [5]: # use a full grid over all parameters
        param_grid = {
            "C": [1e-5, 1e-3, 1e-1, 1],
            "fit_intercept": [True, False],
            "penalty": ["l1", "l2"]
        }

        clf = LogisticRegression()

        # run grid search
        dk_grid_search = GridSearchCV(clf, param_grid=param_grid, n_jobs=-1)
        sk_grid_search = ms.GridSearchCV(clf, param_grid=param_grid, n_jobs=-1)
```

`GridSearchCV` objects are fit just like regular estimators: `.fit(X, y)`.

First, we'll fit the scikit-learn version.

```
In [6]: start = time()
        sk_grid_search.fit(X, y)

        print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
              % (time() - start, len(sk_grid_search.cv_results_['params'])))
```

```
GridSearchCV took 2.93 seconds for 16 candidate parameter settings.
```

And now the `dask-searchcv` version.

```
In [7]: start = time()

        dk_grid_search.fit(X, y)

        print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
              % (time() - start, len(dk_grid_search.cv_results_['params'])))
```

```
GridSearchCV took 1.85 seconds for 16 candidate parameter settings.
```

Randomized Search

Randomized search is similar in spirit to grid search, but the method of choosing parameters to evaluate differs. With grid search, you specify the parameters to try, and scikit-learn tries each possible combination. Randomized search, on the other hand, takes some *distributions to sample from* and a maximum number of iterations to try. This lets you focus your search on areas where the parameters should perform better.

```
In [8]: param_dist = {
            "C": stats.beta(1, 3),
            "fit_intercept": [True, False],
            "penalty": ["l1", "l2"]
        }
        n_iter_search = 100
        clf = LogisticRegression()
```

```
In [9]: # scikit-learn
        sk_random_search = ms.RandomizedSearchCV(clf, param_distributions=param_dist,
                                                 n_iter=n_iter_search, n_jobs=-1)

        # dask
        dk_random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                              n_iter=n_iter_search, n_jobs=-1)
```

```
In [10]: # run randomized search
         start = time()
         sk_random_search.fit(X, y)
```

```
        print("RandomizedSearchCV took %.2f seconds for %d candidates"
              " parameter settings." % ((time() - start), n_iter_search))
```

```
RandomizedSearchCV took 7.64 seconds for 100 candidates parameter settings.
```

```
In [11]: dk_random_search.fit(X, y)
         print("RandomizedSearchCV took %.2f seconds for %d candidates"
               " parameter settings." % ((time() - start), n_iter_search))
```

```
RandomizedSearchCV took 17.11 seconds for 100 candidates parameter settings.
```

Avoid Repeated Work

dask works by building a *task graph* of computations on data. It's able to cache intermediate computations in the graph, to avoid unnecessarily computing something multiple times. This speeds up computations on scikit-learn `Pipelines`, since the early stages of a pipeline are used for each parameter search.

```
In [14]: from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
         from sklearn.linear_model import SGDClassifier
         from sklearn.pipeline import Pipeline

         pipeline = Pipeline([('vect', CountVectorizer()),
                              ('tfidf', TfidfTransformer()),
                              ('clf', SGDClassifier())])

         grid = {'vect__ngram_range': [(1, 1)],
                 'tfidf__norm': ['l1', 'l2'],
                 'clf__alpha': [1e-5, 1e-4, 1e-3, 1e-1]}
```

Using a regular `sklearn.model_selection.GridSearchCV`, we would need to evaluate the `CountVectorizor(ngram_range=(1, 1))` 8 times (once for each of the `tfidf__norm` and `clf__alpha` combintions.

With dask, we need only compute it once and the intermediate result is cached and reused.

```
In [15]: from sklearn.datasets import fetch_20newsgroups

         data = fetch_20newsgroups(subset='train')
```

```
In [16]: sk_grid_search = ms.GridSearchCV(pipeline, grid, n_jobs=-1)
         dk_grid_search = GridSearchCV(pipeline, grid, n_jobs=-1)
```

```
In [17]: start = time()

         dk_grid_search.fit(data.data, data.target)

         print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
               % (time() - start, len(dk_grid_search.cv_results_['params'])))
```

```
GridSearchCV took 34.44 seconds for 8 candidate parameter settings.
```

```
In [18]: start = time()

         sk_grid_search.fit(data.data, data.target)

         print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
               % (time() - start, len(sk_grid_search.cv_results_['params'])))
```

```
GridSearchCV took 40.32 seconds for 8 candidate parameter settings.
```

3.2.2 Distributed Joblib

Scikit-learn already parallelizes many algorithms internally using `joblib`. You can schedule these to run on a distributed cluster using `dask.distributed`, which register's a plugin with `joblib`.

Parts of this example are taken from https://github.com/ogrisel/parallel_ml_tutorial/blob/master/notebooks/06%20-%20Distributed%20Model%20Selection%20and%20Assessment.ipynb

```
In [1]: from time import time
        import matplotlib.pyplot as plt

        from sklearn.datasets import fetch_olivetti_faces
        from sklearn.ensemble import ExtraTreesClassifier
        from sklearn.externals import joblib

        import distributed.joblib
        from distributed import Client
```

As usual, we connect to our client. Typically you would be the address for your scheduler here.

```
In [2]: client = Client()
```

Let's load the data as usual.

```
In [3]: # Load the faces dataset

        data = fetch_olivetti_faces()
        X = data.images.reshape((len(data.images), -1))
        y = data.target

        mask = y < 5  # Limit to 5 classes
        X = X[mask]
        y = y[mask]
```

And create the classifier.

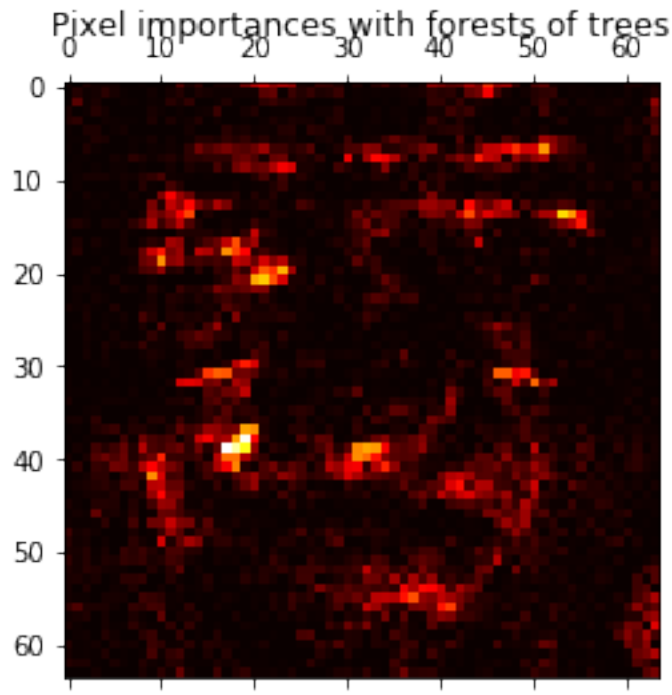
```
In [4]: forest = ExtraTreesClassifier(n_estimators=1000,
                                      max_features=128,
                                      n_jobs=-1,
                                      random_state=0)
```

And now we fit the model. The actual fitting step is the usual `forest.fit(X, y)`. To use our cluster, we'll use the `joblib.parallel_backend` context manager.

```
In [6]: # Build a forest and compute the pixel importances

        t0 = time()
        with joblib.parallel_backend('dask.distributed', scheduler_host=client.scheduler.address):
            forest.fit(X, y)

        print("done in %0.3fs" % (time() - t0))
```

```
done in 1.106s
```

```
In [7]: importances = forest.feature_importances_
        importances = importances.reshape(data.images[0].shape)

        # Plot pixel importances
        plt.matshow(importances, cmap=plt.cm.hot)
        plt.title("Pixel importances with forests of trees")
        plt.show()
```



3.2.3 Out-of-core Prediction

For some estimators, additional data don't improve performance past a certain point. The learning curve levels off. You may have additional data, but using it in the `fit` step won't make any difference.

In these cases, you'll commonly fit a model on a dataset that fits in memory, and use it to predict for datasets that may not. Dask can make the prediction step easier and faster.

```
In [69]: import numpy as np
         import dask.array as da
         from sklearn.datasets import make_classification

In [71]: X_train, y_train = make_classification(
             n_features=2, n_redundant=0, n_informative=2,
             random_state=1, n_clusters_per_class=1, n_samples=1000)

In [72]: N = 100
         X = da.concatenate([da.from_array(X_train, chunks=X_train.shape)
                             for _ in range(N)])
         y = da.concatenate([da.from_array(y_train, chunks=y_train.shape)
                             for _ in range(N)])
```

So `X_train` and `y_train` are regular numpy arrays that we'll use to fit the model. `X` and `y` are large dask arrays that may not fit in memory.

```
In [76]: from sklearn.linear_model import LogisticRegressionCV

In [79]: clf = LogisticRegressionCV()

         clf.fit(X_train, y_train)

Out[79]: LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
                   fit_intercept=True, intercept_scaling=1.0, max_iter=100,
                   multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
                   refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

With the model, we can make predictions for each observation by mapping the `clf.predict_proba` method over each block. This can then be scheduled to run on your single machine or your cluster.

```
In [82]: yhat = X.map_blocks(clf.predict_proba, dtype=np.float64)
         yhat
```

```
Out[82]: dask.array<predict_proba, shape=(100000, 2), dtype=float64, chunksize=(1000, 2)>
```

```
In [83]: yhat[:5].compute()
```

```
Out[83]: array([[ 0.06448766,  0.93551234],
                [ 0.23144553,  0.76855447],
                [ 0.22557832,  0.77442168],
                [ 0.08833674,  0.91166326],
                [ 0.90324265,  0.09675735]])
```

3.2.4 Dask GLM

`dask-glm <https://github.com/dask/dask-glm>`__ is a library for fitting generalized linear models on large datasets. The heart of the project is the set of optimization routines that work on either NumPy or dask arrays. See these two blogposts describing how dask-glm works internally.

This notebook is shows an example of the higher-level scikit-learn style API built on top of these optimization routines.

```
In [1]: import os
        import s3fs
        import pandas as pd
        import dask.array as da
        import dask.dataframe as dd
        from distributed import Client

        from dask import persist, compute
        from dask_glm.estimators import LogisticRegression
```

We'll setup a `distributed.Client <http://distributed.readthedocs.io/en/latest/api.html#distributed.client.Client>`__ locally. In the real world you could connect to a cluster of dask-workers.

```
In [2]: client = Client()
```

For demonstration, we'll use the perennial NYC taxi cab dataset. Since I'm just running things on my laptop, we'll just grab the first month's worth of data.

```
In [4]: ddf = dd.read_csv("trip.csv")
        ddf = ddf.repartition(npartitions=8)
```

I happen to know that some of the values in this dataset are suspect, so let's drop them. Scikit-learn doesn't support filtering observations inside a pipeline (yet), so we'll do this before anything else.

```
In [5]: # these filter out less than 1% of the observations
        ddf = ddf[(ddf.trip_distance < 20) &
                  (ddf.fare_amount < 150)]
        ddf = ddf.repartition(npartitions=8)
```

Now, we'll split our DataFrame into a train and test set, and select our feature matrix and target column (whether the passenger tipped).

```
In [6]: df_train, df_test = ddf.random_split([0.8, 0.2], random_state=2)

        columns = ['VendorID', 'passenger_count', 'trip_distance', 'payment_type', 'fare_amount']

        X_train, y_train = df_train[columns], df_train['tip_amount'] > 0
        X_test, y_test = df_test[columns], df_test['tip_amount'] > 0
```

```
        X_train, y_train, X_test, y_test = persist(
            X_train, y_train, X_test, y_test
        )
```

```
In [7]: X_train.head()
```

Out[7]:

	VendorID	passenger_count	trip_distance	payment_type	fare_amount
2	1	1	1.8	2	9.5
3	1	1	0.5	2	3.5
4	1	1	3.0	2	15.0
5	1	1	9.0	1	27.0
6	1	1	2.2	2	14.0

```
In [8]: y_train.head()
```

```
Out[8]: 2    False
        3    False
        4    False
        5     True
        6    False
        Name: tip_amount, dtype: bool
```

```
In [9]: print(f"{len(X_train):,d} observations")
```

```
10,155,301 observations
```

With our training data in hand, we fit our logistic regression. Nothing here should be surprising to those familiar with `scikit-learn`.

```
In [10]: %%time
         # this is a *dask-glm* LogisticRegresion, not scikit-learn
         lm = LogisticRegression(fit_intercept=False)
         lm.fit(X_train.values, y_train.values)
```

```
CPU times: user 1min 27s, sys: 13.3 s, total: 1min 40s
Wall time: 11min 25s
```

Again, following the lead of scikit-learn we can measure the performance of the estimator on the training dataset using the `.score` method. For LogisticRegression this is the mean accuracy score (what percent of the predicted matched the actual).

```
In [11]: %%time
         lm.score(X_train.values, y_train.values).compute()
```

```
CPU times: user 205 ms, sys: 25 ms, total: 230 ms
Wall time: 364 ms
```

```
Out[11]: 0.88082578743850137
```

and on the test dataset:

```
In [12]: %%time
         lm.score(X_test.values, y_test.values).compute()
```

```
CPU times: user 144 ms, sys: 21.8 ms, total: 166 ms
Wall time: 249 ms
```

```
Out[12]: 0.88061000067744588
```

Pipelines

The bulk of my time "doing data science" is data cleaning and pre-processing. Actually fitting an estimator or making predictions is a relatively small proportion of the work.

You could manually do all your data-processing tasks as a sequence of function calls starting with the raw data. Or, you could use *scikit-learn's ''Pipeline'* <http://scikit-learn.org/stable/modules/pipeline.html>'__ to accomplish this and then some. `Pipelines` offer a few advantages over the manual solution.

First, your entire modeling process from raw data to final output is in a self-contained object. No more wondering "did I remember to scale this version of my model?" It's there in the `Pipeline` for you to check.

Second, `Pipelines` combine well with scikit-learn's model selection utilties, specifically `GridSearchCV` and `RandomizedSearchCV`. You're able to search over hyperparameters of the pipeline stages, just like you would for an estimator.

Third, `Pipelines` help prevent leaking information from your test and validation sets to your training set. A common mistake is to compute some pre-processing statistic on the *entire* dataset (before you've train-test split) rather than just the training set. For example, you might normalize a column by the average of all the observations. These types of errors can lead you overestimate the performance of your model on new observations.

Since dask-glm follows the scikit-learn API, we can reuse scikit-learn's `Pipeline` machinery, *with a few caveats.*

Many of the tranformers built into scikit-learn will validate their inputs. As part of this, array-like things are cast to numpy arrays. Since dask-arrays are array-like they are converted and things "work", but this might not be ideal when your dataset doesn't fit in memory.

Second, some things are just fundamentally hard to do on large datasets. For example, naively dummy-encoding a dataset requires a full scan of the data to determine the set of unique values per categorical column. When your dataset fits in memory, this isn't a huge deal. But when it's scattered across a cluster, this could become a bottleneck.

If you know the set of possible values *ahead* of time, you can do much better. You can encode the categorical columns as pandas `Categoricals`, and then convert with `get_dummies`, without having to do an expensive full-scan, just to compute the set of unique values. We'll do that on the `VendorID` and `payment_type` columnms.

```
In [13]: from sklearn.base import TransformerMixin, BaseEstimator
         from sklearn.pipeline import make_pipeline
```

First let's write a little transformer to convert columns to `Categoricals`. If you aren't familar with scikit-learn transformers, the basic idea is that the transformer must implement two methods: `.fit` and `.tranform`.

`.fit` is called during training. It learns something about the data and records it on `self`.

Then `.transform` uses what's learned during `.fit` to transform the feature matrix somehow.

A `Pipeline` is simply a chain of transformers, each one is `fit` on some data, and passes the output of `.transform` onto the next step; the final output is an `Estimator`, like `LogisticRegression`.

```
In [14]: class CategoricalEncoder(BaseEstimator, TransformerMixin):
             """Encode `categories` as pandas `Categorical`

             Parameters
             ----------
             categories : Dict[str, list]
                 Mapping from column name to list of possible values
             """
             def __init__(self, categories):
                 self.categories = categories

             def fit(self, X, y=None):
                 # "stateless" transformer. Don't have anything to learn here
                 return self

             def transform(self, X, y=None):
                 X = X.copy()
                 for column, categories in self.categories.items():
```

```
                X[column] = X[column].astype('category').cat.set_categories(categories)
        return X
```

We'll also want a daskified version of scikit-learn's `StandardScaler`, that won't eagerly convert a dask.array to a numpy array (N.B. the scikit-learn version has more features and error handling, but this will work for now).

```
In [15]: class StandardScaler(BaseEstimator, TransformerMixin):
             def __init__(self, columns=None, with_mean=True, with_std=True):
                 self.columns = columns
                 self.with_mean = with_mean
                 self.with_std = with_std

             def fit(self, X, y=None):
                 if self.columns is None:
                     self.columns_ = X.columns
                 else:
                     self.columns_ = self.columns
                 if self.with_mean:
                     self.mean_ = X[self.columns_].mean(0)
                 if self.with_std:
                     self.scale_ = X[self.columns_].std(0)
                 return self

             def transform(self, X, y=None):
                 X = X.copy()
                 if self.with_mean:
                     X[self.columns_] = X[self.columns_] - self.mean_
                 if self.with_std:
                     X[self.columns_] = X[self.columns_] / self.scale_
                 return X.values
```

Finally, I've written a dummy encoder transformer that converts categoricals to dummy-encoded interger columns. The full implementation is a bit long for a blog post, but you can see it here.

```
In [16]: from dummy_encoder import DummyEncoder
```

```
In [17]: pipe = make_pipeline(
             CategoricalEncoder({"VendorID": [1, 2],
                                 "payment_type": [1, 2, 3, 4, 5]}),
             DummyEncoder(),
             StandardScaler(columns=['passenger_count', 'trip_distance', 'fare_amount']),
             LogisticRegression(fit_intercept=False)
         )
```

So that's our pipeline. We can go ahead and fit it just like before, passing in the raw data.

```
In [18]: %%time
         pipe.fit(X_train, y_train.values)
```

```
CPU times: user 5min 24s, sys: 42.4 s, total: 6min 6s
Wall time: 37min 4s
```

```
Out[18]: Pipeline(memory=None,
              steps=[('categoricalencoder', CategoricalEncoder(categories={'VendorID': [1, 2], 'payme
                 with_mean=True, ...iter=100, over_relax=1, regularizer='l2', reltol=0.01, rho=1,
                   solver='admm', tol=0.0001))])
```

And we can score it as well. The `Pipeline` ensures that all of the nescessary transformations take place before calling the estimator's `score` method.

```
In [19]: pipe.score(X_train, y_train.values).compute()
```

```
Out[19]: 0.97890756758465358
```

```
In [20]: pipe.score(X_test, y_test.values).compute()
```

```
Out[20]: 0.97888495550125487
```

Grid Search

As explained earlier, Pipelines and grid search go hand-in-hand. Let's run a quick example with dask-searchcv.

```
In [21]: from sklearn.model_selection import GridSearchCV
         import dask_searchcv as dcv
```

We'll search over two hyperparameters

1. Whether or not to standardize the variance of each column in `StandardScaler`
2. The strength of the regularization in `LogisticRegression`

This involves fitting many models, one for each combination of paramters. dask-searchcv is smart enough to know that early stages in the pipeline (like the categorical and dummy encoding) are shared among all the combinations, and so only fits them once.

```
In [22]: param_grid = {
             'standardscaler__with_std': [True, False],
             'logisticregression__lamduh': [.001, .01, .1, 1],
         }

         pipe = make_pipeline(
             CategoricalEncoder({"VendorID": [1, 2],
                                 "payment_type": [1, 2, 3, 4, 5]}),
             DummyEncoder(),
             StandardScaler(columns=['passenger_count', 'trip_distance', 'fare_amount']),
             LogisticRegression(fit_intercept=False)
         )

         gs = dcv.GridSearchCV(pipe, param_grid)
```

```
In [23]: %%time
         gs.fit(X_train, y_train.values)
```

```
CPU times: user 1min 5s, sys: 24 s, total: 1min 29s
Wall time: 40min 24s
```

```
Out[23]: GridSearchCV(cache_cv=True, cv=None, error_score='raise',
                estimator=Pipeline(memory=None,
                steps=[('categoricalencoder', CategoricalEncoder(categories={'VendorID': [1, 2], 'payme
                   with_mean=True, ...iter=100, over_relax=1, regularizer='l2', reltol=0.01, rho=1,
                     solver='admm', tol=0.0001))]),
                iid=True,
                param_grid={'standardscaler__with_std': [True, False], 'logisticregression__lamduh':
                refit=True, return_train_score=True, scoring=None)
```

Now we have access to the usual attributes like `cv_results_` learned by the grid search object:

```
In [31]: pd.DataFrame(gs.cv_results_)
```

```
Out[31]: mean_test_score  mean_train_score param_logisticregression__lamduh  \
         0         0.946754          0.946304                                0.001
         1         0.946754          0.946304                                0.001
         2         0.946754          0.946304                                 0.01
         3         0.946754          0.946304                                 0.01
         4         0.946754          0.946304                                  0.1
         5         0.946754          0.946304                                  0.1
         6         0.946754          0.946304                                    1
```

```
7         0.946754          0.946304                                1

  param_standardscaler__with_std  \
0                           True
1                          False
2                           True
3                          False
4                           True
5                          False
6                           True
7                          False

                                              params  rank_test_score  \
0  {'logisticregression__lamduh': 0.001, 'standar...                1
1  {'logisticregression__lamduh': 0.001, 'standar...                1
2  {'logisticregression__lamduh': 0.01, 'standard...                1
3  {'logisticregression__lamduh': 0.01, 'standard...                1
4  {'logisticregression__lamduh': 0.1, 'standards...                1
5  {'logisticregression__lamduh': 0.1, 'standards...                1
6  {'logisticregression__lamduh': 1, 'standardsca...                1
7  {'logisticregression__lamduh': 1, 'standardsca...                1

   split0_test_score  split0_train_score  split1_test_score  \
0           0.978945            0.978819           0.882557
1           0.978945            0.978819           0.882557
2           0.978945            0.978819           0.882557
3           0.978945            0.978819           0.882557
4           0.978945            0.978819           0.882557
5           0.978945            0.978819           0.882557
6           0.978945            0.978819           0.882557
7           0.978945            0.978819           0.882557

   split1_train_score  split2_test_score  split2_train_score  std_test_score  \
0            0.881177           0.978758            0.978918        0.045394
1            0.881177           0.978758            0.978918        0.045394
2            0.881177           0.978758            0.978918        0.045394
3            0.881177           0.978758            0.978918        0.045394
4            0.881177           0.978758            0.978918        0.045394
5            0.881177           0.978758            0.978918        0.045394
6            0.881177           0.978758            0.978918        0.045394
7            0.881177           0.978758            0.978918        0.045394

   std_train_score
0         0.046052
1         0.046052
2         0.046052
3         0.046052
4         0.046052
5         0.046052
6         0.046052
7         0.046052
```

And we can do our usual checks on model fit for the training set:

```
In [25]: gs.score(X_train, y_train.values).compute()
```

```
Out[25]: 0.97888698720008394
```

And the test set:

```
In [26]: gs.score(X_test, y_test.values).compute()
```

```
Out[26]: 0.97886801935289736
```

Hopefully your reaction to everything here is somewhere between a nodding head and a yawn. If you're familiar with scikit-learn, everything here should look pretty routine. It's the same API you know and love, scaled out to larger datasets thanks to dask-glm.

3.2.5 Dask and XGBoost

The `dask-xgboost` library provides a small wrapper around `dask-xgboost` for passing dask objects to `xgboost`.

As usual, we can create or attach to our cluster by creating a `Client`

```
In [2]: import os
        from dask import compute, persist
        from dask.distributed import Client, progress

        client = Client(os.environ.get("DISTRIBUTED_ADDRESS"))  # connect to cluster
```

We'll work through an example using the airlines dataset. For a recorded screen cast, see here. Our task is to pick whether or not a flight was delayed for more than 15 minutes.

First, let's load the data.

```
In [3]: import dask.dataframe as dd

        # Subset of the columns to use
        cols = ['Year', 'Month', 'DayOfWeek', 'Distance',
                'DepDelay', 'CRSDepTime', 'UniqueCarrier', 'Origin', 'Dest']

        # Create the dataframe
        df = dd.read_csv('s3://dask-data/airline-data/2006.csv', usecols=cols,
                         storage_options={'anon': True})

        frac = 0.01
        df = df.sample(frac=frac) # we blow out ram otherwise

        is_delayed = (df.DepDelay.fillna(16) > 15)

        df['CRSDepTime'] = df['CRSDepTime'].clip(upper=2399)
        del df['DepDelay']

        df, is_delayed = persist(df, is_delayed)
        progress(df, is_delayed)
```

```
In [6]: df.head()
```

```
Out[6]:
        Year  Month  DayOfWeek  CRSDepTime UniqueCarrier Origin Dest  Distance
156561  2006      1          2        1440            OH    CVG  MHT       741
359225  2006      1          5        1241            MQ    DFW  CLE      1021
414374  2006      1          6        1031            NW    MBS  DTW        98
39002   2006      1          3         725            WN    CRP  HOU       187
488494  2006      1          3        1259            AA    DFW  BDL      1471
```

```
In [7]: is_delayed.head()
```

```
Out[7]:
156561    False
359225    False
414374    False
39002     False
```

```
        488494    False
        Name: DepDelay, dtype: bool

In [8]: df2 = dd.get_dummies(df.categorize()).persist()
```

Split and Train

We'll split the original dataset into train and test sets. The model will be fit on `data_train` and evaluated on `data_test`.

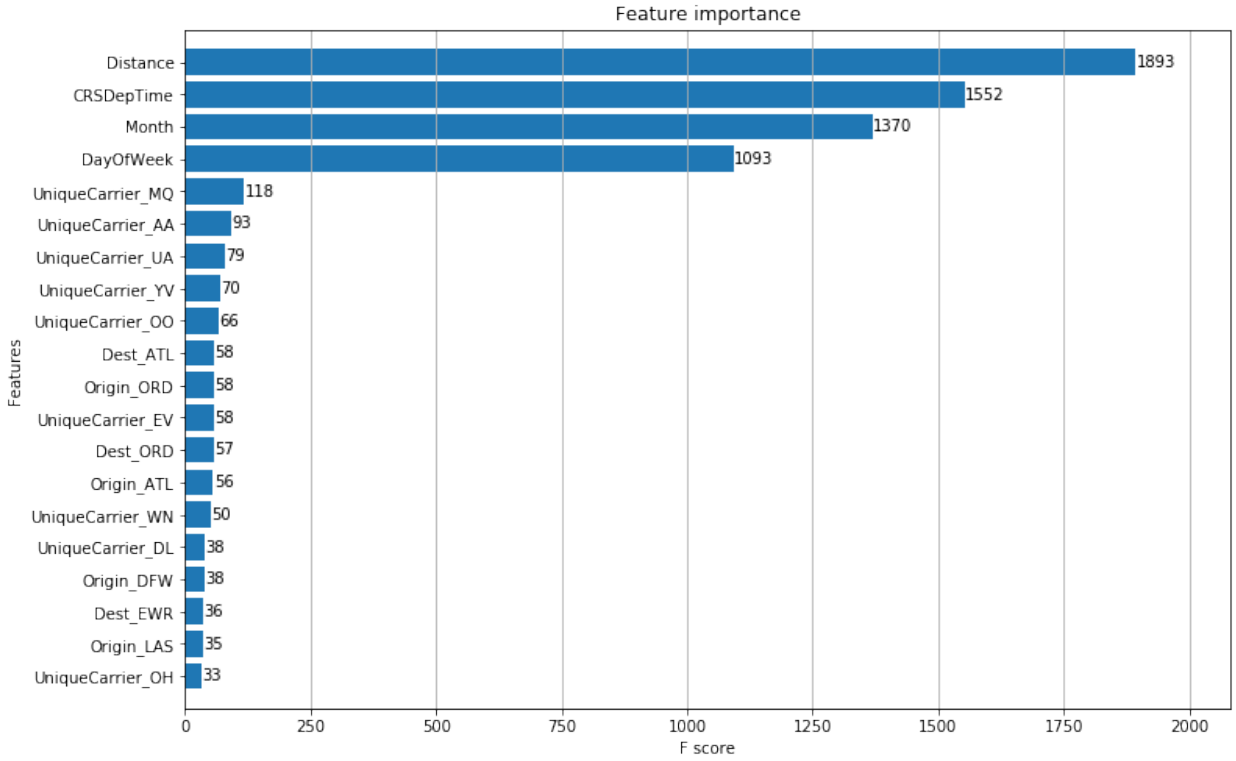
```
In [9]: data_train, data_test = df2.random_split([0.9, 0.1],
                                                 random_state=1234)
        labels_train, labels_test = is_delayed.random_split([0.9, 0.1],
                                                            random_state=1234)
```

Now we can fit the model. This is exactly like normal, except you'll import `dask_xgboost`. It will take care of handing the data off to the distributed `xgboost` processes, and from there everything proceeds as normal.

```
In [10]: %%time
         import dask_xgboost as dxgb
         import xgboost as xgb

         params = {'objective': 'binary:logistic', 'nround': 1000,
                   'max_depth': 16, 'eta': 0.01, 'subsample': 0.5,
                   'min_child_weight': 1}

         bst = dxgb.train(client, params, data_train, labels_train)
         bst

CPU times: user 962 ms, sys: 569 ms, total: 1.53 s
Wall time: 17.2 s
```

The `bst` object is just a reguler `xgboost.Booster`, so all the familar methods are available.

```
In [11]: import matplotlib.pyplot as plt

In [12]: fig, ax = plt.subplots(figsize=(12, 8))

         ax = xgb.plot_importance(bst, ax=ax, height=0.8, max_num_features=20)
         ax.grid("off", axis="y")
```



Let's get the predictions. The is the same as usual, except we're using `dxgb`.

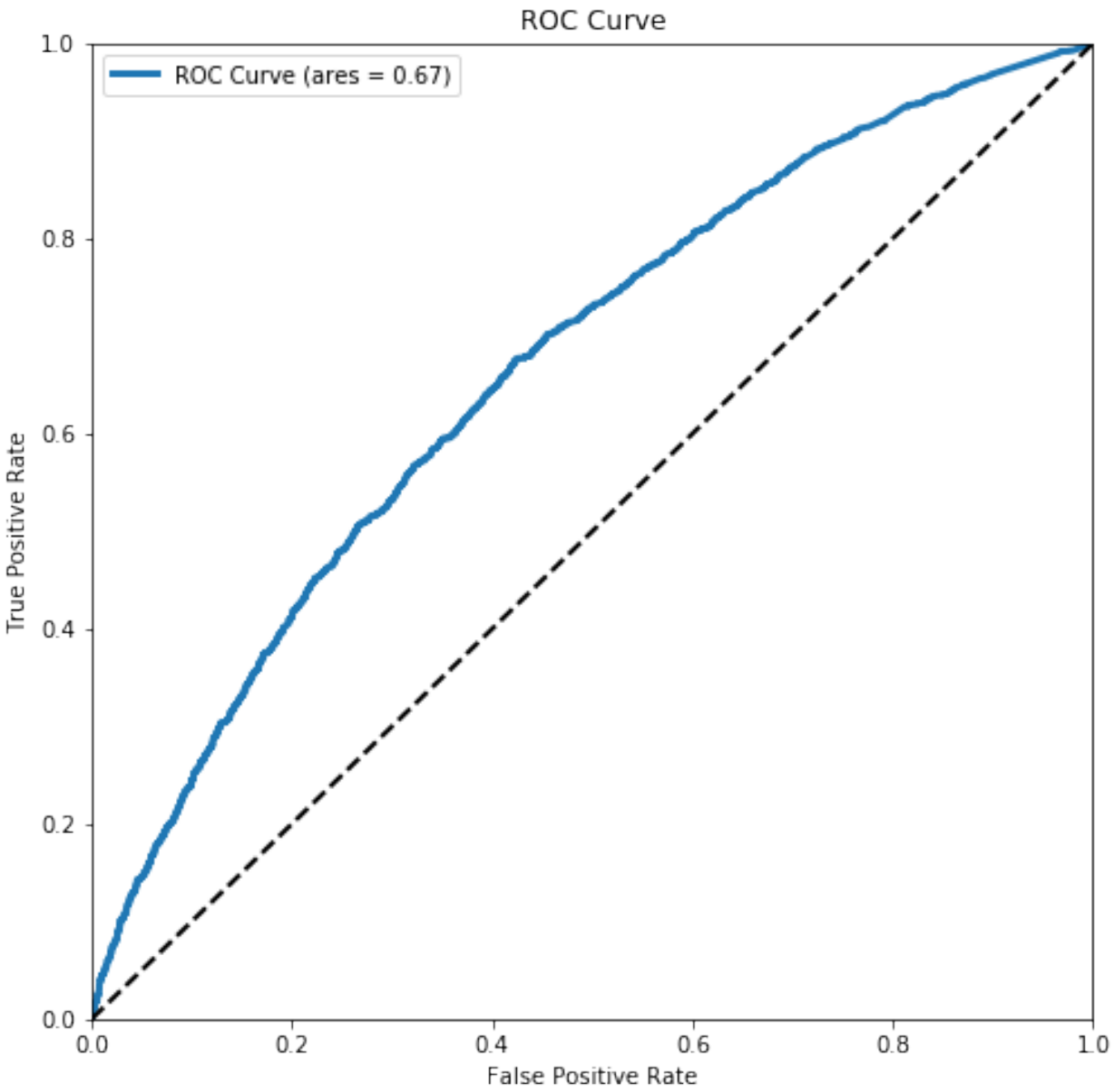
```
In [13]: predictions = dxgb.predict(client, bst, data_test)
         predictions = client.persist(predictions)
         predictions
```

```
Out[13]: Dask Series Structure:
         npartitions=11
         None    float32
         None        ...
                  ...
         None        ...
         None        ...
         Name: predictions, dtype: float32
         Dask Name: _predict_part, 11 tasks
```

The `predictions` object is a `dask.Series`, so it can be used in place of a numpy array in most places.

```
In [14]: from sklearn.metrics import roc_curve, auc
```

```
In [15]: fig, ax = plt.subplots(figsize=(8, 8))
         fpr, tpr, _ = roc_curve(labels_test, predictions)
         ax.plot(fpr, tpr, lw=3,
                 label='ROC Curve (ares = {:.2f})'.format(auc(fpr, tpr)))
         ax.plot([0, 1], [0, 1], 'k--', lw=2)

         ax.set(
             xlim=(0, 1),
             ylim=(0, 1),
             title="ROC Curve",
             xlabel="False Positive Rate",
             ylabel="True Positive Rate",
         )
         ax.legend();
```



3.2.6 Dask and Tensorflow

See http://matthewrocklin.com/blog/work/2017/02/11/dask-tensorflow for more.

```
In [2]: %matplotlib inline
```

```
In [1]: import dask.array as da
        from dask import delayed
        from dask_tensorflow import start_tensorflow
        from distributed import Client, progress, get_worker, worker_client, Queue
        import dask.dataframe as dd
        import matplotlib.pyplot as plt
        import tensorflow as tf
```

```
In [3]: client = Client()
```

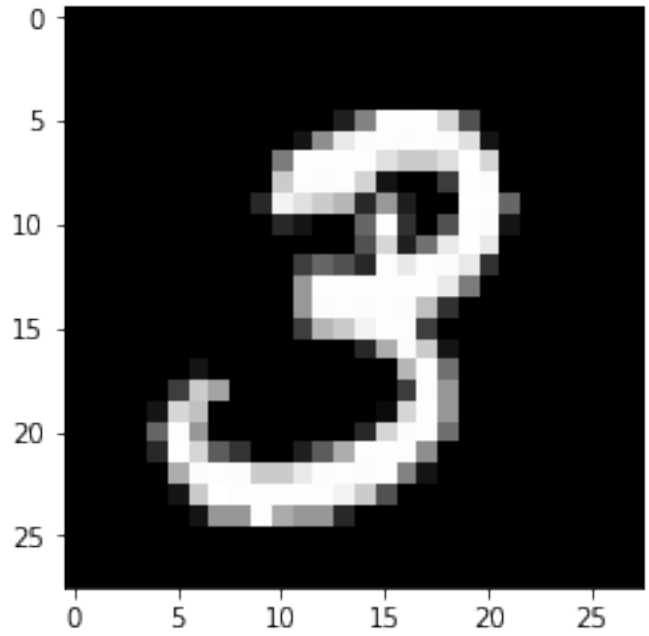
```
In [4]: def get_mnist():
            from tensorflow.examples.tutorials.mnist import input_data
            mnist = input_data.read_data_sets('/tmp/mnist-data', one_hot=True)
            return mnist.train.images, mnist.train.labels

        datasets = [delayed(get_mnist)() for i in range(20)]  # 20 versions of same dataset
        images = [d[0] for d in datasets]
        labels = [d[1] for d in datasets]

        images = [da.from_delayed(im, shape=(55000, 784), dtype='float32') for im in images]
        labels = [da.from_delayed(la, shape=(55000, 10), dtype='float32') for la in labels]

        images = da.concatenate(images, axis=0)
        labels = da.concatenate(labels, axis=0)

        images
```

```
Out[4]: dask.array<concatenate, shape=(1100000, 784), dtype=float32, chunksize=(55000, 784)>
```

```
In [5]: images, labels = client.persist([images, labels])
```

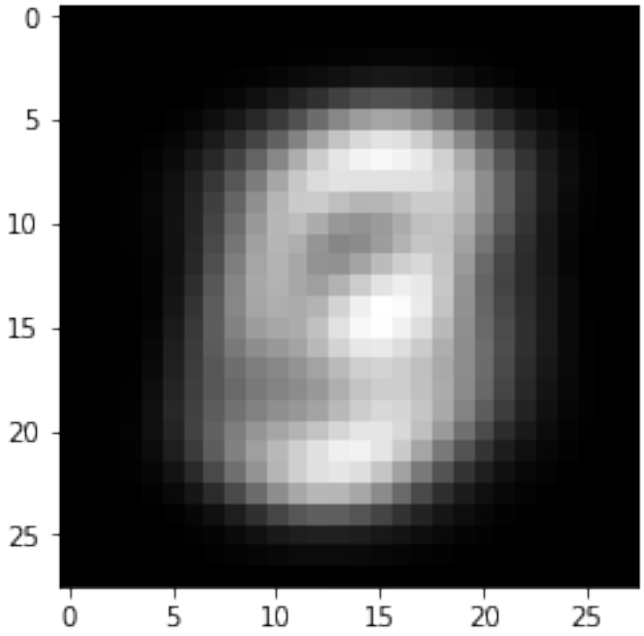
```
In [7]: im = images[1].compute().reshape((28, 28))
        plt.imshow(im, cmap='gray')
```

```
Out[7]: <matplotlib.image.AxesImage at 0x117ce19e8>
```




```
In [8]: im = images.mean(axis=0).compute().reshape((28, 28))
        plt.imshow(im, cmap='gray')
```

```
Out[8]: <matplotlib.image.AxesImage at 0x117d2d668>
```




```
In [9]: images = images.rechunk((10000, 784))
        labels = labels.rechunk((10000, 10))

        images = images.to_delayed().flatten().tolist()
        labels = labels.to_delayed().flatten().tolist()
        batches = [delayed([im, la]) for im, la in zip(images, labels)]

        batches = client.compute(batches)
```

```
In [11]: tf_spec, dask_spec = start_tensorflow(client, ps=1, worker=4, scorer=1)
```

```
In [13]: import math
         import tempfile
         import time
         from queue import Empty

         IMAGE_PIXELS = 28
         hidden_units = 100
         learning_rate = 0.01
         sync_replicas = False
         replicas_to_aggregate = len(dask_spec['worker'])
```

```
In [14]: def model(server):
             worker_device = "/job:%s/task:%d" % (server.server_def.job_name,
                                                  server.server_def.task_index)
             task_index = server.server_def.task_index
             is_chief = task_index == 0

             with tf.device(tf.train.replica_device_setter(
                                worker_device=worker_device,
                                ps_device="/job:ps/cpu:0",
                                cluster=tf_spec)):

                 global_step = tf.Variable(0, name="global_step", trainable=False)

                 # Variables of the hidden layer
```

```
hid_w = tf.Variable(
    tf.truncated_normal(
        [IMAGE_PIXELS * IMAGE_PIXELS, hidden_units],
        stddev=1.0 / IMAGE_PIXELS),
    name="hid_w")
hid_b = tf.Variable(tf.zeros([hidden_units]), name="hid_b")

# Variables of the softmax layer
sm_w = tf.Variable(
    tf.truncated_normal(
        [hidden_units, 10],
        stddev=1.0 / math.sqrt(hidden_units)),
    name="sm_w")
sm_b = tf.Variable(tf.zeros([10]), name="sm_b")

# Ops: located on the worker specified with task_index
x = tf.placeholder(tf.float32, [None, IMAGE_PIXELS * IMAGE_PIXELS])
y_ = tf.placeholder(tf.float32, [None, 10])

hid_lin = tf.nn.xw_plus_b(x, hid_w, hid_b)
hid = tf.nn.relu(hid_lin)

y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b))
cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))

opt = tf.train.AdamOptimizer(learning_rate)

if sync_replicas:
    if replicas_to_aggregate is None:
        replicas_to_aggregate = num_workers
    else:
        replicas_to_aggregate = replicas_to_aggregate

    opt = tf.train.SyncReplicasOptimizer(
              opt,
              replicas_to_aggregate=replicas_to_aggregate,
              total_num_replicas=num_workers,
              name="mnist_sync_replicas")

train_step = opt.minimize(cross_entropy, global_step=global_step)

if sync_replicas:
    local_init_op = opt.local_step_init_op
    if is_chief:
        local_init_op = opt.chief_init_op

    ready_for_local_init_op = opt.ready_for_local_init_op

    # Initial token and chief queue runners required by the sync_replicas mode
    chief_queue_runner = opt.get_chief_queue_runner()
    sync_init_op = opt.get_init_tokens_op()

init_op = tf.global_variables_initializer()
train_dir = tempfile.mkdtemp()

if sync_replicas:
  sv = tf.train.Supervisor(
      is_chief=is_chief,
      logdir=train_dir,
```

```
                        init_op=init_op,
                        local_init_op=local_init_op,
                        ready_for_local_init_op=ready_for_local_init_op,
                        recovery_wait_secs=1,
                        global_step=global_step)
                else:
                  sv = tf.train.Supervisor(
                        is_chief=is_chief,
                        logdir=train_dir,
                        init_op=init_op,
                        recovery_wait_secs=1,
                        global_step=global_step)

                sess_config = tf.ConfigProto(
                    allow_soft_placement=True,
                    log_device_placement=False,
                    device_filters=["/job:ps", "/job:worker/task:%d" % task_index])

                # The chief worker (task_index==0) session will prepare the session,
                # while the remaining workers will wait for the preparation to complete.
                if is_chief:
                  print("Worker %d: Initializing session..." % task_index)
                else:
                  print("Worker %d: Waiting for session to be initialized..." %
                        task_index)

                sess = sv.prepare_or_wait_for_session(server.target, config=sess_config)

                if sync_replicas and is_chief:
                  # Chief worker will start the chief queue runner and call the init op.
                  sess.run(sync_init_op)
                  sv.start_queue_runners(sess, [chief_queue_runner])

                return sess, x, y_, train_step, global_step, cross_entropy
In [17]: def ps_task():
             worker = get_worker()
             worker.tensorflow_server.join()
In [18]: def scoring_task():
             with worker_client() as c:
                 # Scores Channel
                 scores = Queue('scores')

                 # Make Model
                 worker = get_worker()
                 server = worker.tensorflow_server
                 sess, x, y_, _, _, cross_entropy = model(server)

                 # Testing Data
                 from tensorflow.examples.tutorials.mnist import input_data
                 mnist = input_data.read_data_sets('/tmp/mnist-data', one_hot=True)
                 test_data = {x: mnist.validation.images,
                              y_: mnist.validation.labels}

                 # Main Loop
                 while True:
                     score = sess.run(cross_entropy, feed_dict=test_data)
                     scores.put(float(score))
```

```
                    time.sleep(1)

In [19]: def worker_task():
             with worker_client() as c:
                 scores = Queue('scores')
                 num_workers = replicas_to_aggregate = len(dask_spec['worker'])

                 worker = get_worker()
                 server = worker.tensorflow_server
                 tf_queue = worker.tensorflow_queue

                 # Make model
                 sess, x, y_, train_step, global_step, _= model(server)

                 # Main loop
                 sc = scores.get()
                 while not scores or sc > 1000:
                     try:
                         batch = tf_queue.get(timeout=0.5)
                     except Empty:
                         continue

                     train_data = {x: batch[0],
                                   y_: batch[1]}

                     sess.run([train_step, global_step], feed_dict=train_data)

In [21]: ps_tasks = [client.submit(ps_task, workers=worker)
                     for worker in dask_spec['ps']]

         worker_tasks = [client.submit(worker_task, workers=addr, pure=False)
                         for addr in dask_spec['worker']]

         scorer_task = client.submit(scoring_task, workers=dask_spec['scorer'][0])

In [ ]: def transfer_dask_to_tensorflow(batch):
            worker = get_worker()
            worker.tensorflow_queue.put(batch)

In [23]: scores = Queue('scores')
         score = scores.get()
         while score > 1000:
             print(score)
             dump = client.map(transfer_dask_to_tensorflow, batches,
                               workers=dask_spec['worker'], pure=False)
             score = scores.get()
```

We also have sets of

- examples in the dask-examples repository, which has machine learning examples that can be launched online with Binder.
- Dask-ML performance benchmarks in the dask-ml-benchmarks repository

3.3 Preprocessing

dask_ml.preprocessing contains some scikit-learn style transformers that can be used in `Pipelines` to perform various data transformations as part of the model fitting process. These transformers will work well on dask

collections (`dask.array`, `dask.dataframe`), NumPy arrays, or pandas dataframes. They'll fit and transform in parallel.

3.3.1 Scikit-Learn Clones

Some of the transformers are (mostly) drop-in replacements for their scikit-learn counterparts.

MinMaxScaler([feature_range, copy])	Transforms features by scaling each feature to a given range.
QuantileTransformer([n_quantiles, ...])	Transforms features using quantile information.
RobustScaler([with_centering, with_scaling, ...])	Scale features using statistics that are robust to outliers.
StandardScaler([copy, with_mean, with_std])	Standardize features by removing the mean and scaling to unit variance
LabelEncoder	Encode labels with value between 0 and n_classes-1.

These can be used just like the scikit-learn versions, except that:

1. They operate on dask collections in parallel
2. `.transform` will return a `dask.array` or `dask.dataframe` when the input is a dask collection

See `sklearn.preprocessing` for more information about any particular transformer.

3.3.2 Additional Tranformers

Other transformers are specific to dask-ml.

Categorizer([categories, columns])	Transform columns of a DataFrame to categorical dtype.
DummyEncoder([columns, drop_first])	Dummy (one-hot) encode categorical columns.
OrdinalEncoder([columns])	Ordinal (integer) encode categorical columns.

Both *dask_ml.preprocessing.Categorizer* and *dask_ml.preprocessing.DummyEncoder* deal with converting non-numeric data to numeric data. They are useful as a preprocessing step in a pipeline where you start with heterogenous data (a mix of numeric and non-numeric), but the estimator requires all numeric data.

In this toy example, we use a dataset with two columns. `'A'` is numeric and `'B'` contains text data. We make a small pipeline to

1. Categorize the text data
2. Dummy encode the categorical data
3. Fit a linear regression

```
In [1]: from dask_ml.preprocessing import Categorizer, DummyEncoder

In [2]: from sklearn.linear_model import LogisticRegression

In [3]: from sklearn.pipeline import make_pipeline

In [4]: import pandas as pd

In [5]: import dask.dataframe as dd
```

(continues on next page)

(continued from previous page)

```
In [6]: df = pd.DataFrame({"A": [1, 2, 1, 2], "B": ["a", "b", "c", "c"]})

In [7]: X = dd.from_pandas(df, npartitions=2)

In [8]: y = dd.from_pandas(pd.Series([0, 1, 1, 0]), npartitions=2)

In [9]: pipe = make_pipeline(
   ...:     Categorizer(),
   ...:     DummyEncoder(),
   ...:     LogisticRegression()
   ...: )
   ...:

In [10]: pipe.fit(X, y)
Out[10]:
Pipeline(memory=None,
     steps=[('categorizer', Categorizer(categories=None, columns=None)), (
↪'dummyencoder', DummyEncoder(columns=None, drop_first=False)), ('logisticregression
↪', LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))])
```

`Categorizer` will convert a subset of the columns in `X` to categorical dtype (see here for more about how pandas handles categorical data). By default, it converts all the `object` dtype columns.

`DummyEncoder` will dummy (or one-hot) encode the dataset. This replaces a categorical column with multiple columns, where the values are either 0 or 1, depending on whether the value in the original.

```
In [11]: df['B']
Out[11]:
0    a
1    b
2    c
3    c
Name: B, dtype: object

In [12]: pd.get_dummies(df['B'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  0  0  1
```

Wherever the original was `'a'`, the transformed now has a `1` in the `a` column and a `0` everywhere else.

Why was the `Categorizizer` step necessary? Why couldn't we operate directly on the `object` (string) dtype column? Doing this would be fragile, especially when using `dask.dataframe`, since *the shape of the output would depend on the values present*. For example, suppose that we just saw the first two rows in the training, and the last two rows in the tests datasets. Then, when training, our transformed columns would be:

```
In [13]: pd.get_dummies(df.loc[[0, 1], 'B'])
Out[13]:
   a  b
0  1  0
1  0  1
```

while on the test dataset, they would be:

```
In [14]: pd.get_dummies(df.loc[[2, 3], 'B'])
Out[14]:
   c
2  1
3  1
```

Which is incorrect! The columns don't match.

When we categorize the data, we can be confident that all the possible values have been specified, so the output shape no longer depends on the values in the whatever subset of the data we currently see. Instead, it depends on the `categories`, which are identical in all the subsets.

3.4 Cross Validation

See the scikit-learn cross validation documentation for a fuller discussion of cross validation. This document only describes the extensions made to support Dask arrays.

The simplest way to split one or more Dask arrays is with `dask_ml.model_selection.train_test_split()`.

```
In [1]: import dask.array as da

In [2]: from dask_ml.datasets import make_regression

In [3]: from dask_ml.model_selection import train_test_split

In [4]: X, y = make_regression(n_samples=125, n_features=4, random_state=0, chunks=50)

In [5]: X
Out[5]: dask.array<normal, shape=(125, 4), dtype=float64, chunksize=(50, 4)>
```

The interface for splitting Dask arrays is the same as scikit-learn's version.

```
In [6]: X_train, X_test, y_train, y_test = train_test_split(X, y)

In [7]: X_train  # A dask Array
Out[7]: dask.array<concatenate, shape=(112, 4), dtype=float64, chunksize=(45, 4)>

In [8]: X_train.compute()[:3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]:␣
↪
array([[-1.34564068,  1.35118465,  0.48875575,  1.04930275],
       [-1.32835409,  0.3065724 ,  0.43893223,  0.65025546],
       [-0.3122028 , -0.88191003, -0.00495866,  0.91761886]])
```

While it's possible to pass dask arrays to `sklearn.model_selection.train_test_split()`, we recommend using the Dask version for performance reasons. There are two major difference that let make the Dask version faster.

First, **the Dask version shuffles blockwise**. In a distributed setting, shuffling *between* blocks may require sending large amounts of data between machines, which can be slow. However, if there's a strong pattern in your data, you'll want to perform a full shuffle.

Second, the Dask version avoids allocating large intermediate NumPy arrays storing the indexes for slicing. For very large datasets, creating and transmitting `np.arange(n_samples)` can be expensive.

3.5 Hyper Parameter Search

Tools for performing hyperparameter optimization of Scikit-Learn API-compatible models using Dask. Dask-ML implements GridSearchCV and RandomizedSearchCV.

`sklearn.model_selection.GridSearchCV(...[, ...])`	Exhaustive search over specified parameter values for an estimator.
`dask_ml.model_selection.GridSearchCV(...[, ...])`	Exhaustive search over specified parameter values for an estimator.
`sklearn.model_selection.RandomizedSearchCV(...)`	Randomized search on hyper parameters.
`dask_ml.model_selection.RandomizedSearchCV(...)`	Randomized search on hyper parameters.

The varians in Dask-ML implement many (but not all) of the same parameters, and should be a drop-in replacement for the subset that they do implement. In that case, why use Dask-ML's versions?

- *Flexible Backends*: Hyperparameter optimization can be done in parallel using threads, processes, or distributed across a cluster.
- *Works well with Dask collections*. Dask arrays, dataframes, and delayed can be passed to `fit`.
- *Avoid repeated work*. Candidate estimators with identical parameters and inputs will only be fit once. For composite-estimators such as `Pipeline` this can be significantly more efficient as it can avoid expensive repeated computations.

Both scikit-learn's and Dask-ML's model selection meta-estimators can be used with Dask's *joblib backend*.

3.5.1 Flexible Backends

Dask-searchcv can use any of the dask schedulers. By default the threaded scheduler is used, but this can easily be swapped out for the multiprocessing or distributed scheduler:

```
# Distribute grid-search across a cluster
from dask.distributed import Client
scheduler_address = '127.0.0.1:8786'
client = Client(scheduler_address)

search.fit(digits.data, digits.target)
```

3.5.2 Works Well With Dask Collections

Dask collections such as `dask.array`, `dask.dataframe` and `dask.delayed` can be passed to `fit`. This means you can use dask to do your data loading and preprocessing as well, allowing for a clean workflow. This also allows you to work with remote data on a cluster without ever having to pull it locally to your computer:

```
import dask.dataframe as dd

# Load data from s3
df = dd.read_csv('s3://bucket-name/my-data-*.csv')

# Do some preprocessing steps
df['x2'] = df.x - df.x.mean()
```

(continues on next page)

(continued from previous page)

```
# ...

# Pass to fit without ever leaving the cluster
search.fit(df[['x', 'x2']], df['y'])
```

3.5.3 Avoid Repeated Work

When searching over composite estimators like `sklearn.pipeline.Pipeline` or `sklearn.pipeline.FeatureUnion`, Dask-ML will avoid fitting the same estimator + parameter + data combination more than once. For pipelines with expensive early steps this can be faster, as repeated work is avoided.

For example, given the following 3-stage pipeline and grid (modified from this scikit-learn example).

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline

pipeline = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', SGDClassifier())])

grid = {'vect__ngram_range': [(1, 1)],
        'tfidf__norm': ['l1', 'l2'],
        'clf__alpha': [1e-3, 1e-4, 1e-5]}
```

the Scikit-Learn grid-search implementation looks something like (simplified):

```
scores = []
for ngram_range in parameters['vect__ngram_range']:
        for norm in parameters['tfidf__norm']:
                for alpha in parameters['clf__alpha']:
                        vect = CountVectorizer(ngram_range=ngram_range)
                        X2 = vect.fit_transform(X, y)
                        tfidf = TfidfTransformer(norm=norm)
                        X3 = tfidf.fit_transform(X2, y)
                        clf = SGDClassifier(alpha=alpha)
                        clf.fit(X3, y)
                        scores.append(clf.score(X3, y))
best = choose_best_parameters(scores, parameters)
```

As a directed acyclic graph, this might look like:

In contrast, the dask version looks more like:

```
scores = []
for ngram_range in parameters['vect__ngram_range']:
        vect = CountVectorizer(ngram_range=ngram_range)
        X2 = vect.fit_transform(X, y)
        for norm in parameters['tfidf__norm']:
                tfidf = TfidfTransformer(norm=norm)
                X3 = tfidf.fit_transform(X2, y)
                for alpha in parameters['clf__alpha']:
                        clf = SGDClassifier(alpha=alpha)
                        clf.fit(X3, y)
```

(continues on next page)

(continued from previous page)

```
                        scores.append(clf.score(X3, y))
best = choose_best_parameters(scores, parameters)
```

With a corresponding directed acyclic graph:

Looking closely, you can see that the Scikit-Learn version ends up fitting earlier steps in the pipeline multiple times with the same parameters and data. Due to the increased flexibility of Dask over Joblib, we're able to merge these tasks in the graph and only perform the fit step once for any parameter/data/estimator combination. For pipelines that have relatively expensive early steps, this can be a big win when performing a grid search.

Pipelines

Dask-ML uses scikit-learn's `sklearn.pipeline.Pipeline` to express pipelines of estimators that are chained together. If the individual estimators work well with Dask's collections, the pipeline will as well.

3.6 Generalized Linear Models

LinearRegression([penalty, dual, tol, C, ...])	Esimator for linear_regression.
LogisticRegression([penalty, dual, tol, C, ...])	Esimator for logistic_regression.
PoissonRegression([penalty, dual, tol, C, ...])	Esimator for poisson_regression.

Generalized linear models are a broad class of commonly used models. These implementations scale well out to large datasets either on a single machine or distributed cluster. They can be powered by a variety of optimization algorithms and use a variety of regularizers.

These follow the scikit-learn estimator API, and so can be dropped into existing routines like grid search and pipelines, but are implemented externally with new, scalable algorithms and so can consume distributed dask arrays and dataframes rather than just single-machine NumPy and Pandas arrays and dataframes.

3.6.1 Example

```
In [1]: from dask_ml.linear_model import LogisticRegression

In [2]: from dask_ml.datasets import make_classification

In [3]: X, y = make_classification(chunks=50)

In [4]: lr = LogisticRegression()

In [5]: lr.fit(X, y)
Out[5]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1.0, max_iter=100, multiclass='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='admm',
          solver_kwargs=None, tol=0.0001, verbose=0, warm_start=False)
```

3.6.2 Algorithms

admm(X, y[, regularizer, lamduh, rho, ...])	Alternating Direction Method of Multipliers
gradient_descent(X, y[, max_iter, tol, family])	Michael Grant's implementation of Gradient Descent.
lbfgs(X, y[, regularizer, lamduh, max_iter, ...])	L-BFGS solver using scipy.optimize implementation
newton(X, y[, max_iter, tol, family])	Newtons Method for Logistic Regression.
proximal_grad(X, y[, regularizer, lamduh, ...])	

Parameters

3.6.3 Regularizers

ElasticNet([weight])	Elastic net regularization.
L1	L1 regularization.
L2	L2 regularization.
Regularizer	Abstract base class for regularization object.

3.7 Joblib

Dask.distributed integrates with Joblib by providing an alternative cluster-computing backend, alongside Joblib's builtin threading and multiprocessing backends. This enables training a scikit-learn model in parallel using a cluster of machines.

The following video demonstrates how to use Dask to parallelize a grid search across a cluster.

Joblib is a library for simple parallel programming primarily developed and used by the Scikit Learn community. As of version 0.10.0 it contains a plugin mechanism to allow Joblib code to use other parallel frameworks to execute computations. The `dask.distributed` scheduler implements such a plugin in the `dask_ml.joblib` module and registers it appropriately with Joblib when imported. As a result, any joblib code (including many scikit-learn algorithms) will run on the distributed scheduler if you enclose it in a context manager as follows:

```
import dask_ml.joblib  # registers joblib plugin
from dask.distributed import Client

client = Client(processes=False)              # create local cluster
# client = Client("scheduler-address:8786")  # or connect to remote cluster

from joblib import Parallel, parallel_backend

with parallel_backend('dask'):
    # normal Joblib code
```

Note that scikit-learn bundles joblib internally, so if you want to specify the joblib backend you'll need to import `parallel_backend` from scikit-learn instead of `joblib`. As an example you might distributed a randomized cross validated parameter search as follows.

```
import dask_ml.joblib  # registers joblib plugin
# Scikit-learn bundles joblib, so you need to import from
# `sklearn.externals.joblib` instead of `joblib` directly
from sklearn.externals.joblib import parallel_backend
from sklearn.datasets import load_digits
from sklearn.grid_search import RandomizedSearchCV
```

(continues on next page)

(continued from previous page)

```
from sklearn.svm import SVC
import numpy as np

digits = load_digits()

param_space = {
    'C': np.logspace(-6, 6, 13),
    'gamma': np.logspace(-8, 8, 17),
    'tol': np.logspace(-4, -1, 4),
    'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=3, n_iter=50, verbose=10)

with parallel_backend('dask'):
    search.fit(digits.data, digits.target)
```

For large arguments that are used by multiple tasks, it may be more efficient to pre-scatter the data to every worker, rather than serializing it once for every task. This can be done using the `scatter` keyword argument, which takes an iterable of objects to send to each worker.

```
# Serialize the training data only once to each worker
with parallel_backend('dask', scatter=[digits.data, digits.target]):
    search.fit(digits.data, digits.target)
```

3.8 Parallel Meta-estimators

dask-ml provides some meta-estimators that parallelize and scaling out certain tasks that may not be parallelized within scikit-learn itself. For example, *ParallelPostFit* will parallelize the `predict`, `predict_proba` and `transform` methods, enabling them to work on large (possibly larger-than memory) datasets.

3.8.1 Parallel Prediction and Transformation

wrappers.ParallelPostFit is a meta-estimator for parallelizing post-fit tasks like prediction and transformation. It can wrap any scikit-learn estimator to provide parallel `predict`, `predict_proba`, and `transform` methods.

Warning: `ParallelPostFit` does *not* parallelize the training step. The underlying estimator's `.fit` method is called normally.

Since just the `predict`, `predict_proba`, and `transform` methods are wrapped, *wrappers.ParallelPostFit* is most useful in situations where your training dataset is relatively small (fits in a single machine's memory), and prediction or transformation must be done on a much larger dataset (perhaps larger than a single machine's memory).

```
In [1]: from sklearn.ensemble import GradientBoostingClassifier

In [2]: import sklearn.datasets
```

(continues on next page)

(continued from previous page)

```
In [3]: import dask_ml.datasets

In [4]: from dask_ml.wrappers import ParallelPostFit
```

In this example, we'll make a small 1,000 sample training dataset

```
In [5]: X, y = sklearn.datasets.make_classification(n_samples=1000,
   ...:                                            random_state=0)
   ...:
```

Training is identical to just calling `estimator.fit(X, y)`. Aside from copying over learned attributes, that's all that `ParallelPostFit` does.

```
In [6]: clf = ParallelPostFit(estimator=GradientBoostingClassifier())

In [7]: clf.fit(X, y)
Out[7]:
ParallelPostFit(estimator=GradientBoostingClassifier(criterion='friedman_mse',␣
↪init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=None, subsample=1.0, verbose=0,
              warm_start=False),
        scoring=None)
```

This class is useful for predicting for or transforming large datasets. We'll make a larger dask array `X_big` with 10,000 samples per block.

```
In [8]: X_big, _ = dask_ml.datasets.make_classification(n_samples=100000,
   ...:                                                 chunks=10000,
   ...:                                                 random_state=0)
   ...:

In [9]: clf.predict(X_big)
Out[9]: dask.array<predict, shape=(100000,), dtype=int64, chunksize=(10000,)>
```

This returned a `dask.array`. Like any dask array, the actual `compute` will cause the scheduler to compute tasks in parallel. If you've connected to a `dask.distributed.Client`, the computation will be parallelized across your cluster of machines.

```
In [10]: clf.predict_proba(X_big).compute()[:10]
Out[10]:
array([[0.24519991, 0.75480009],
       [0.00464304, 0.99535696],
       [0.00902734, 0.99097266],
       [0.01299259, 0.98700741],
       [0.9287794 , 0.0712206 ],
       [0.755881  , 0.244119  ],
       [0.8938643 , 0.1061357 ],
       [0.0203811 , 0.9796189 ],
       [0.021179  , 0.978821  ],
       [0.97064062, 0.02935938]])
```

See parallelizing prediction for an example of how this scales for a support vector classifier.

3.8.2 Comparison to other Estimators in dask-ml

`dask-ml` re-implements some estimators from scikit-learn, for example *dask_ml.cluster.KMeans*, or *dask_ml.preprocessing.QuantileTransformer*. This raises the question, should I use the reimplemented dask-ml versions, or should I wrap scikit-learn version in a meta-estimator? It varies from estimator to estimator, and depends on your tolerance for approximate solutions and the size of your training data. In general, if your training data is small, you should be fine wrapping the scikit-learn version with a `dask-ml` meta-estimator.

3.9 Incremental Learning

Some estimators can be trained incrementally – without seeing the entire dataset at once. Scikit-Learn provdes the `partial_fit` API to stream batches of data to an estimator that can be fit in batches.

Normally, if you pass a Dask Array to an estimator expecting a NumPy array, the Dask Array will be converted to a single, large NumPy array. On a single machine, you'll likely run out of RAM and crash the program. On a distributed cluster, all the workers will send their data to a single machine and crash it.

dask_ml.wrappers.Incremental provides a bridge between Dask and Scikit-Learn estimators supporting the `partial_fit` API. You wrap the underlying estimator in `Incremental`. Dask-ML will sequentially pass each block of a Dask Array to the underlying estimator's `partial_fit` method.

3.9.1 Incremental Meta-estimator

wrappers.Incremental([estimator, scoring, ...])	Metaestimator for feeding Dask Arrays to an estimator blockwise.

dask_ml.wrappers.Incremental is a meta-estimator (an estimator that takes another estimator) that bridges scikit-learn estimators expecting NumPy arrays, and users with large Dask Arrays.

Each *block* of a Dask Array is fed to the underlying estiamtor's `partial_fit` method. The training is entirely sequential, so you won't notice massive training time speedups from parallelism. In a distributed environment, you should notice some speedup from avoiding extra IO, and the fact that models are typically much smaller than data, and so faster to move between machines.

```
In [1]: from dask_ml.datasets import make_classification

In [2]: from dask_ml.wrappers import Incremental

In [3]: from sklearn.linear_model import SGDClassifier

In [4]: X, y = make_classification(chunks=25)

In [5]: X
Out[5]: dask.array<normal, shape=(100, 20), dtype=float64, chunksize=(25, 20)>

In [6]: estimator = SGDClassifier(random_state=10, max_iter=100)

In [7]: clf = Incremental(estimator)

In [8]: clf.fit(X, y, classes=[0, 1])
Out[8]:
Incremental(estimator=SGDClassifier(alpha=0.0001, average=False, class_weight=None,␣
↪epsilon=0.1,
```

(continues on next page)

(continued from previous page)

```
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=100, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=10, shuffle=True,
       tol=None, verbose=0, warm_start=False),
      random_state=None, scoring=None, shuffle_blocks=True)
```

In this example, we make a (small) random Dask Array. It has 100 samples, broken in the 4 blocks of 25 samples each. The chunking is only along the first axis (the samples). There is no chunking along the features.

You instantiate the underlying estimator as usual. It really is just a scikit-learn compatible estimator, and will be trained normally via its `partial_fit`.

Notice that we call the regular `.fit` method, not `partial_fit` for training. Dask-ML takes care of passing each block to the underlying estimator for you.

Just like `sklearn.linear_model.SGDClassifier.partial_fit()`, we need to pass the `classes` argument to `fit`. In general, any argument that is required for the underlying estimators `parital_fit` becomes required for the wrapped `fit`.

Note: Take care with the behavior of `Incremental.score()`. Most estimators inherit the default scoring methods of R2 score for regressors and accuracy score for classifiers. For these estimators, we automatically use Dask-ML's scoring methods, which are able to operate on Dask arrays.

If your underlying estimator uses a different scoring method, you'll need to ensure that the scoring method is able to operate on Dask arrays. You can also explicitly pass `scoring=` to pass a dask-aware scorer.

We can get the accuracy score on our dataset.

```
In [9]: clf.score(X, y)
Out[9]: 0.64
```

All of the attributes learned durning training, like `coef_`, are available on the `Incremental` instance.

```
In [10]: clf.coef_
Out[10]:
array([[-18.64155936, -61.99137144, -19.01326946,  -5.91318918,
         21.70499718, -18.26138418, -18.14378713, -39.76024612,
         15.54996869,   3.47654649,   7.35780488,  -0.4861231 ,
         13.54228649,  29.71163633, -56.76564906,  45.07381123,
        -12.45314251,  -3.22397579,  -9.32686924,  28.9940955 ]])
```

If necessary, the actual estimator trained is available as `Incremental.estimator_`

```
In [11]: clf.estimator_
Out[11]:
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=100, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=10, shuffle=True,
       tol=None, verbose=0, warm_start=False)
```

Incremental Learning and Hyper-parameter Optimization

`Incremental` is a meta-estimator. To search over the hyper-parameters of the underlying estimator, use the usual scikit-learn convention of prefixing the parameter name with `<name>__`. For `Incremental`, `name` is always

estimator.

```
In [12]: from sklearn.model_selection import GridSearchCV

In [13]: param_grid = {'estimator__alpha': [0.10, 10.0]}

In [14]: gs = GridSearchCV(clf, param_grid)

In [15]: gs.fit(X, y, classes=[0, 1])
Out[15]:
GridSearchCV(cv=None, error_score='raise',
       estimator=Incremental(estimator=SGDClassifier(alpha=0.0001, average=False,␣
↪class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=100, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=10, shuffle=True,
       tol=None, verbose=0, warm_start=False),
      random_state=None, scoring=None, shuffle_blocks=True),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'estimator__alpha': [0.1, 10.0]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

This can be mixed with *Joblib* to use a cluster for training in parallel, even if you're RAM-bound.

3.10 Clustering

KMeans([n_clusters, init, ...])	Scalable KMeans for clustering
PartialMiniBatchKMeans(**kwargs)	Mini-Batch K-Means clustering
SpectralClustering([n_clusters, ...])	Apply parallel Spectral Clustering

The *dask_ml.cluster* module implements several algorithms for clustering unlabeled data.

3.10.1 Spectral Clustering

Spectral Clustering finds a low-dimensional embedding on the affinity matrix between samples. The embedded dataset is then clustered, typically with KMeans.

Typically, spectral clustering algorithms do not scale well. Computing the $n\_samples \times n\_samples$ affinity matrix becomes prohibitively expensive when the number of samples is large. Several algorithms have been proposed to work around this limitation.

In `dask-ml`, we use the Nystrom method to approximate the large affinity matrix. This involves sampling `n_components` rows from the entire training set. The exact affinity is computed for this subset ($n\_components \times n\_components$), and between this small subset and the rest of the data ($n\_components \times (n\_samples - n\_components)$). We avoid the direct computation of the rest of the affinity matrix.

Let S be our $n \times n$ affinity matrix. We can rewrite that as

$$S_d = \begin{bmatrix} A \\ B \\ B^T \\ C \end{bmatrix}$$

Where A is the $n \times n$ affinity matrix of the $n\_components$ that we sampled, and B is the $n \times (n - n\_components)$ affinity matrix between the sample and the rest of the dataset. Instead of computing C directly, we approximate it with $B^T A^{-1} B$.

See the spectral clustering benchmark for an example showing how *dask_ml.cluster.SpectralClustering* scales in the number of samples.

3.11 XGBoost

train(client, params, data, labels[, ...])	Train an XGBoost model on a Dask Cluster
predict(client, model, data)	Distributed prediction with XGBoost
XGBClassifier([max_depth, learning_rate, ...])	**Attributes**
XGBRegressor([max_depth, learning_rate, ...])	**Attributes**

XGBoost is a powerful and popular library for gradient boosted trees. For larger datasets or faster training XGBoost also provides a distributed computing solution. Dask-ML can set up distributed XGBoost for you and hand off data from distributed dask.dataframes. This automates much of the hassle of preprocessing and setup while still letting XGBoost do what it does well.

3.11.1 Example

```
from dask.distributed import Client
client = Client('scheduler-address:8786')

import dask.dataframe as dd
df = dd.read_parquet('s3://...')

# Split into training and testing data
train, test = df.random_split([0.8, 0.2])

# Separate labels from data
train_labels = train.x > 0
test_labels = test.x > 0

del train['x']  # remove informative column from data
del test['x']  # remove informative column from data

# from xgboost import XGBRegressor  # change import
from dask_ml.xgboost import XGBRegressor

est = XGBRegressor(...)
est.fit(train, train_labels)

prediction = est.predict(test)
```

3.11.2 How this works

Dask sets up XGBoost's master process on the Dask scheduler and XGBoost's worker processes on Dask's worker processes. Then it moves all of the Dask dataframes' constituent Pandas dataframes to XGBoost and lets XGBoost train. Fortunately, because XGBoost has an excellent Python interface, all of this can happen in the same process without any data transfer. The two distributed services can operate together on the same data.

When XGBoost is finished training Dask cleans up the XGBoost infrastructure and continues on as normal.

This work was a collaboration with XGBoost and SKLearn maintainers. See relevant GitHub issue here: dmlc/xgboost #2032

- xgboost

3.12 Tensorflow

`start_tensorflow`

Tensorflow is a library for numerical computation that's commonly used in deep learning. It can be run in a distributed mode, and `start_tensorflow()` aids in setting up the Tensorflow cluster along side your existing dask cluster.

3.12.1 Example

Given a Dask cluster

```
from dask.distributed import Client
client = Client('scheduler-address:8786')
```

Get a TensorFlow cluster, specifying groups by name

```
from dask_tensorflow import start_tensorflow
tf_spec, dask_spec = start_tensorflow(client, ps=2, worker=4)

>>> tf_spec
{'worker': ['192.168.1.100:2222', '192.168.1.101:2222',
            '192.168.1.102:2222', '192.168.1.103:2222'],
 'ps': ['192.168.1.104:2222', '192.168.1.105:2222']}
```

This creates a `tensorflow.train.Server` on each Dask worker and sets up a Queue for data transfer on each worker. These are accessible directly as `tensorflow_server` and `tensorflow_queue` attributes on the workers.

3.12.2 More Complex Workflow

Typically then we set up long running Dask tasks that get these servers and participate in general TensorFlow computations.

```
from dask.distributed import worker_client

def ps_function(self):
    with worker_client() as c:
        tf_server = c.worker.tensorflow_server
```

(continues on next page)

(continued from previous page)

```
        tf_server.join()

ps_tasks = [client.submit(ps_function, workers=worker, pure=False)
            for worker in dask_spec['ps']]

def worker_function(self):
    with worker_client() as c:
        tf_server = c.worker.tensorflow_server

        # ... use tensorflow as desired ...

worker_tasks = [client.submit(worker_function, workers=worker, pure=False)
                for worker in dask_spec['worker']]
```

One simple and flexible approach is to have these functions block on queues and feed them data from dask arrays, dataframes, etc.

```
def worker_function(self):
    with worker_client() as c:
        tf_server = c.worker.tensorflow_server
        queue = c.worker.tensorflow_queue

        while not stopping_condition():
            batch = queue.get()
            # train with batch
```

And then dump blocks of numpy and pandas dataframes to these queues

```
from distributed.worker_client import get_worker
def dump_batch(batch):
    worker = get_worker()
    worker.tensorflow_queue.put(batch)


import dask.dataframe as dd
df = dd.read_csv('hdfs:///path/to/*.csv')
# clean up dataframe as necessary
partitions = df.to_delayed()  # delayed pandas dataframes
client.map(dump_batch, partitions)
```

3.13 API Reference

This page lists all of the estimators and top-level functions in `dask_ml`. Unless otherwise noted, the estimators implemented in `dask-ml` are appropriate for parallel and distributed training.

3.13.1 `dask_ml.model_selection`: Model Selection

Utilities for hyperparameter optimization.

These estimators will operate in parallel. Their scalability depends on the underlying estimators being used.

Dask-ML has a few cross validation utilities.

model_selection.train_test_split(*arrays, ...)	Split arrays into random train and test matricies.

dask_ml.model_selection.train_test_split

dask_ml.model_selection.**train_test_split**(**arrays*, ***options*)

Split arrays into random train and test matricies.

Parameters

***arrays** [Sequence of Dask Arrays]

test_size [float or int, defualt 0.1]

train_size: float or int, optional

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

shuffle [bool, default True] Whether to shuffle the data before splitting.

blockwise [bool, default True] Whether to perform blockwise-shuffling.

Returns

splitting [list, length=2 * len(arrays)] List containing train-test split of inputs

Examples

import dask.array as da from dask_ml.datasets import make_regression

```
>>> X, y = make_regression(n_samples=125, n_features=4, chunks=50,
...                        random_state=0)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=0)
>>> X_train
dask.array<concatenate, shape=(113, 4), dtype=float64, chunksize=(45, 4)>
>>> X_train.compute()[:2]
array([[ 0.12372191,  0.58222459,  0.92950511, -2.09460307],
       [ 0.99439439, -0.70972797, -0.27567053,  1.73887268]])
```

model_selection.train_test_split() is a simple helper that uses *model_selection.ShuffleSplit* internally.

model_selection.ShuffleSplit([n_splits, ...])	Random permutation cross-validator.

dask_ml.model_selection.ShuffleSplit

class dask_ml.model_selection.**ShuffleSplit**(*n_splits=10*, *test_size=0.1*, *train_size=None*, *blockwise=True*, *random_state=None*)

Random permutation cross-validator.

Yields indices to split data into training and test sets.

Warning: By default, this performs a blockwise-shuffle. That is, each block is shuffled internally, but data are not shuffled between blocks. If your data is ordered, then set `blockwise=False`.

Note: contrary to other cross-validation strategies, random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Parameters

n_splits [int, default 10] Number of re-shuffling & splitting iterations.

test_size [float, int, None, default=0.1] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size.

train_size [float, int, or None, default=None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

blockwise [bool, default True] Whether to shuffle data only within blocks (True), or allow data to be shuffled between blocks (False). Shuffling between blocks can be much more expensive, especially in distributed environments.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Methods

get_n_splits([X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(X[, y, groups])	Generate indices to split data into training and test set.

__init__(*n_splits=10*, *test_size=0.1*, *train_size=None*, *blockwise=True*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

get_n_splits(*X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

split(*X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Returns

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Dask-ML provides drop-in replacements for grid and randomized search.

model_selection.GridSearchCV(estimator, ...)	Exhaustive search over specified parameter values for an estimator.
model_selection.RandomizedSearchCV(...[, ...])	Randomized search on hyper parameters.

dask_ml.model_selection.GridSearchCV

class dask_ml.model_selection.**GridSearchCV**(*estimator*, *param_grid*, *scoring=None*, *iid=True*, *refit=True*, *cv=None*, *error_score='raise'*, *return_train_score='warn'*, *scheduler=None*, *n_jobs=-1*, *cache_cv=True*)

Exhaustive search over specified parameter values for an estimator.

GridSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Parameters

estimator [estimator object.] This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid [dict or list of dictionaries] Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scoring [string, callable, list/tuple, dict or None, default: None] A single string or a callable to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

If None, the estimator's default scorer (if available) is used.

iid [boolean, default=True] If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- An object to be used as a cross-validation generator.
- An iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

refit [boolean, or string, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer is used to find the best parameters for refitting the estimator at the end.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GridSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_parameters_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

See `scoring` parameter to know more about multiple metric evaluation.

error_score ['raise' (default) or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

return_train_score [boolean, default=True] If `'False'`, the `cv_results_` attribute will not include training scores.

Note that for scikit-learn >= 0.19.1, the default of `True` is deprecated, and a warning will be raised when accessing train score results without explicitly asking for train scores.

scheduler [string, callable, Client, or None, default=None] The dask scheduler to use. Default is to use the global scheduler if set, and fallback to the threaded scheduler otherwise. To use a different scheduler either specify it by name (either "threading", "multiprocessing", or "synchronous"), pass in a `dask.distributed.Client`, or provide a scheduler `get` function.

n_jobs [int, default=-1] Number of jobs to run in parallel. Ignored for the synchronous and distributed schedulers. If `n_jobs == -1` [default] all cpus are used. For `n_jobs < -1`, `(n_cpus + 1 + n_jobs)` are used.

cache_cv [bool, default=True] Whether to extract each train/test subset at most once in each worker process, or every time that subset is needed. Caching the splits can speedup computation at the cost of increased memory usage per worker process.

If True, worst case memory usage is `(n_splits + 1) * (X.nbytes + y.nbytes)` per worker. If False, worst case memory usage is `(n_threads_per_worker + 1) * (X.nbytes + y.nbytes)` per worker.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	param_degree	split0_test_score	...	rank....
'poly'	–	2	0.8	...	2
'poly'	–	3	0.7	...	4
'rbf'	0.1	–	0.8	...	3
'rbf'	0.2	–	0.9	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...
↪)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...
↪),
'split0_test_score'  : [0.8, 0.7, 0.8, 0.9],
'split1_test_score'  : [0.82, 0.5, 0.7, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.82],
'std_test_score'     : [0.02, 0.01, 0.03, 0.03],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.8, 0.7, 0.8, 0.9],
'split1_train_score' : [0.82, 0.7, 0.82, 0.5],
'mean_train_score'   : [0.81, 0.7, 0.81, 0.7],
'std_train_score'    : [0.03, 0.04, 0.03, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.007, 0.06, 0.04, 0.04],
'std_score_time'     : [0.001, 0.002, 0.003, 0.005],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

NOTE that the key `'params'` is used to store a list of parameter settings dict for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

best_estimator_ [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if refit=False.

best_score_ [float or dict of floats] Score of best_estimator on the left out data. When using multiple metrics, `best_score_` will be a dictionary where the keys are the names of the scorers, and the values are the mean test score for that scorer.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

best_index_ [int or dict of ints] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

When using multiple metrics, `best_index_` will be a dictionary where the keys are the names of the scorers, and the values are the index with the best mean score for that scorer, as described above.

scorer_ [function or dict of functions] Scorer function used on the held out data to choose the best parameters for the model. A dictionary of `{scorer_name: scorer}` when multiple metrics are used.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

Examples

```
>>> import dask_ml.model_selection as dcv
>>> from sklearn import svm, datasets
>>> iris = datasets.load_iris()
>>> parameters = {'kernel': ['linear', 'rbf'], 'C': [1, 10]}
>>> svc = svm.SVC()
>>> clf = dcv.GridSearchCV(svc, parameters)
>>> clf.fit(iris.data, iris.target)
GridSearchCV(cache_cv=..., cv=..., error_score=...,
        estimator=SVC(C=..., cache_size=..., class_weight=..., coef0=...,
                      decision_function_shape=..., degree=..., gamma=...,
                      kernel=..., max_iter=-1, probability=False,
                      random_state=..., shrinking=..., tol=...,
                      verbose=...),
        iid=..., n_jobs=..., param_grid=..., refit=..., return_train_score=...,
        scheduler=..., scoring=...)
>>> sorted(clf.cv_results_.keys())
['mean_fit_time', 'mean_score_time', 'mean_test_score',...
 'mean_train_score', 'param_C', 'param_kernel', 'params',...
 'rank_test_score', 'split0_test_score',...
 'split0_train_score', 'split1_test_score', 'split1_train_score',...
 'split2_test_score', 'split2_train_score',...
 'std_fit_time', 'std_score_time', 'std_test_score', 'std_train_score'...]
```

Methods

decision_function(X)	Call decision_function on the estimator with the best found parameters.
fit(X[, y, groups])	Run fit with all sets of parameters.
get_params([deep])	Get parameters for this estimator.
inverse_transform(Xt)	Call inverse_transform on the estimator with the best found params.
predict(X)	Call predict on the estimator with the best found parameters.
predict_log_proba(X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(X)	Call predict_proba on the estimator with the best found parameters.

Continued on next page

Table 15 – continued from previous page

score(X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Call transform on the estimator with the best found parameters.
visualize([filename, format])	Render the task graph for this parameter search using graphviz.

__init__ (*estimator*, *param_grid*, *scoring=None*, *iid=True*, *refit=True*, *cv=None*, *error_score='raise'*, *return_train_score='warn'*, *scheduler=None*, *n_jobs=-1*, *cache_cv=True*)
Initialize self. See help(type(self)) for accurate signature.

decision_function (*X*)
Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit (*X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, shape = [n_samples], optional] Group labels for the samples used while splitting the dataset into train/test set.

****fit_params** Parameters passed to the `fit` method of the estimator

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*Xt*)
Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict (*X*)
Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

score(*X*, *y=None*)

Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

visualize(*filename='mydask'*, *format=None*, ***kwargs*)
Render the task graph for this parameter search using `graphviz`.

Requires `graphviz` to be installed.

Parameters

filename [str or None, optional] The name (without an extension) of the file to write to disk. If *filename* is None, no file will be written, and we communicate with dot using only pipes.

format [{'png', 'pdf', 'dot', 'svg', 'jpeg', 'jpg'}, optional] Format in which to write output file. Default is 'png'.

****kwargs** Additional keyword arguments to forward to `dask.dot.to_graphviz`.

Returns

result [IPython.diplay.Image, IPython.display.SVG, or None] See `dask.dot.dot_graph` for more information.

dask_ml.model_selection.RandomizedSearchCV

class dask_ml.model_selection.**RandomizedSearchCV**(*estimator*, *param_distributions*, *n_iter=10*, *random_state=None*, *scoring=None*, *iid=True*, *refit=True*, *cv=None*, *error_score='raise'*, *return_train_score='warn'*, *scheduler=None*, *n_jobs=-1*, *cache_cv=True*)
Randomized search on hyper parameters.

RandomizedSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Parameters

estimator [estimator object.] This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_distributions [dict] Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a `rvs` method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly.

n_iter [int, default=10] Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

random_state [int or RandomState] Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions.

scoring [string, callable, list/tuple, dict or None, default: None] A single string or a callable to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

If None, the estimator's default scorer (if available) is used.

iid [boolean, default=True] If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- An object to be used as a cross-validation generator.
- An iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

refit [boolean, or string, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer is used to find the best parameters for refitting the estimator at the end.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GridSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_parameters_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

See `scoring` parameter to know more about multiple metric evaluation.

error_score ['raise' (default) or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error.

return_train_score [boolean, default=True] If `'False'`, the `cv_results_` attribute will not include training scores.

Note that for scikit-learn >= 0.19.1, the default of `True` is deprecated, and a warning will be raised when accessing train score results without explicitly asking for train scores.

scheduler [string, callable, Client, or None, default=None] The dask scheduler to use. Default is to use the global scheduler if set, and fallback to the threaded scheduler otherwise. To use a different scheduler either specify it by name (either "threading", "multiprocessing", or "synchronous"), pass in a `dask.distributed.Client`, or provide a scheduler `get` function.

n_jobs [int, default=-1] Number of jobs to run in parallel. Ignored for the synchronous and distributed schedulers. If `n_jobs == -1` [default] all cpus are used. For `n_jobs < -1`, `(n_cpus + 1 + n_jobs)` are used.

cache_cv [bool, default=True] Whether to extract each train/test subset at most once in each worker process, or every time that subset is needed. Caching the splits can speedup computation at the cost of increased memory usage per worker process.

If True, worst case memory usage is `(n_splits + 1) * (X.nbytes + y.nbytes)` per worker. If False, worst case memory usage is `(n_threads_per_worker + 1) * (X.nbytes + y.nbytes)` per worker.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	param_degree	split0_test_score	...	rank.....
'poly'	–	2	0.8	...	2
'poly'	–	3	0.7	...	4
'rbf'	0.1	–	0.8	...	3
'rbf'	0.2	–	0.9	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...
↪)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...
↪),
'split0_test_score'  : [0.8, 0.7, 0.8, 0.9],
'split1_test_score'  : [0.82, 0.5, 0.7, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.82],
'std_test_score'     : [0.02, 0.01, 0.03, 0.03],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.8, 0.7, 0.8, 0.9],
'split1_train_score' : [0.82, 0.7, 0.82, 0.5],
'mean_train_score'   : [0.81, 0.7, 0.81, 0.7],
'std_train_score'    : [0.03, 0.04, 0.03, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.007, 0.06, 0.04, 0.04],
'std_score_time'     : [0.001, 0.002, 0.003, 0.005],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

NOTE that the key `'params'` is used to store a list of parameter settings dict for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

best_estimator_ [estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if refit=False.

best_score_ [float or dict of floats] Score of best_estimator on the left out data. When using multiple metrics, `best_score_` will be a dictionary where the keys are the names of the scorers, and the values are the mean test score for that scorer.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

best_index_ [int or dict of ints] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

When using multiple metrics, `best_index_` will be a dictionary where the keys are the names of the scorers, and the values are the index with the best mean score for that scorer, as described above.

scorer_ [function or dict of functions] Scorer function used on the held out data to choose the best parameters for the model. A dictionary of `{scorer_name: scorer}` when multiple metrics are used.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

Examples

```
>>> import dask_ml.model_selection as dcv
>>> from scipy.stats import expon
>>> from sklearn import svm, datasets
>>> iris = datasets.load_iris()
>>> parameters = {'C': expon(scale=100), 'kernel': ['linear', 'rbf']}
>>> svc = svm.SVC()
>>> clf = dcv.RandomizedSearchCV(svc, parameters, n_iter=100)
>>> clf.fit(iris.data, iris.target)
RandomizedSearchCV(cache_cv=..., cv=..., error_score=...,
        estimator=SVC(C=..., cache_size=..., class_weight=..., coef0=...,
                      decision_function_shape=..., degree=..., gamma=...,
                      kernel=..., max_iter=..., probability=...,
                      random_state=..., shrinking=..., tol=...,
                      verbose=...),
        iid=..., n_iter=..., n_jobs=..., param_distributions=...,
        random_state=..., refit=..., return_train_score=...,
        scheduler=..., scoring=...)
>>> sorted(clf.cv_results_.keys())
['mean_fit_time', 'mean_score_time', 'mean_test_score',...
 'mean_train_score', 'param_C', 'param_kernel', 'params',...
 'rank_test_score', 'split0_test_score',...
 'split0_train_score', 'split1_test_score', 'split1_train_score',...
 'split2_test_score', 'split2_train_score',...
 'std_fit_time', 'std_score_time', 'std_test_score', 'std_train_score'...]
```

Methods

decision_function(X)	Call decision_function on the estimator with the best found parameters.
fit(X[, y, groups])	Run fit with all sets of parameters.
get_params([deep])	Get parameters for this estimator.
inverse_transform(Xt)	Call inverse_transform on the estimator with the best found params.
predict(X)	Call predict on the estimator with the best found parameters.
predict_log_proba(X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(X)	Call predict_proba on the estimator with the best found parameters.
score(X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Call transform on the estimator with the best found parameters.
visualize([filename, format])	Render the task graph for this parameter search using `graphviz`.

__init__(*estimator*, *param_distributions*, *n_iter=10*, *random_state=None*, *scoring=None*, *iid=True*, *refit=True*, *cv=None*, *error_score='raise'*, *return_train_score='warn'*, *scheduler=None*, *n_jobs=-1*, *cache_cv=True*)

Initialize self. See help(type(self)) for accurate signature.

decision_function(*X*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit(*X*, *y=None*, *groups=None*, ***fit_params*)

Run fit with all sets of parameters.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vector, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, shape = [n_samples], optional] Group labels for the samples used while splitting the dataset into train/test set.

****fit_params** Parameters passed to the `fit` method of the estimator

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*Xt*)

Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict(*X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

score(*X*, *y=None*)

Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

visualize(*filename='mydask'*, *format=None*, ***kwargs*)

Render the task graph for this parameter search using `graphviz`.

Requires `graphviz` to be installed.

Parameters

filename [str or None, optional] The name (without an extension) of the file to write to disk. If *filename* is None, no file will be written, and we communicate with dot using only pipes.

format [{'png', 'pdf', 'dot', 'svg', 'jpeg', 'jpg'}, optional] Format in which to write output file. Default is 'png'.

****kwargs** Additional keyword arguments to forward to `dask.dot.to_graphviz`.

Returns

result [IPython.diplay.Image, IPython.display.SVG, or None] See `dask.dot.dot_graph` for more information.

3.13.2 `dask_ml.linear_model`: Generalized Linear Models

The `dask_ml.linear_model` module implements linear models for classification and regression.

linear_model.LinearRegression([penalty, ...])	Esimator for linear_regression.
linear_model.LogisticRegression([penalty, ...])	Esimator for logistic_regression.
linear_model.PoissonRegression([penalty, ...])	Esimator for poisson_regression.

dask_ml.linear_model.LinearRegression

class dask_ml.linear_model.**LinearRegression**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)

Esimator for linear_regression.

Parameters

penalty [str or Regularizer, default 'l2'] Regularizer to use. Only relevant for the 'admm',

'lbfgs' and 'proximal_grad' solvers.

For string values, only 'l1' or 'l2' are valid.

dual [bool] Ignored

tol [float, default 1e-4] The tolerance for convergence.

C [float] Regularization strength. Note that `dask-glm` solvers use the parameterization $\lambda = 1/C$

fit_intercept [bool, default True] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

intercept_scaling [bool] Ignored

class_weight [dict or 'balanced'] Ignored

random_state [int, RandomState, or None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when solver == 'sag' or 'liblinear'.

solver [{'admm', 'gradient_descent', 'newton', 'lbfgs', 'proximal_grad'}] Solver to use. See Algorithms for details

multiclass [str, default 'ovr'] Ignored. Multiclass solvers not currently supported.

verbose [int, default 0] Ignored

warm_start [bool, default False] Ignored

n_jobs [int, default 1] Ignored

solver_kwargs [dict, optional, default None] Extra keyword arguments to pass through to the solver.

Attributes

coef_ [array, shape (n_classes, n_features)] The learned value for the model's coefficients

intercept_ [float of None] The learned value for the intercept, if one was added to the model

Examples

```
>>> from dask_glm.datasets import make_regression
>>> X, y = make_regression()
>>> lr = LinearRegression()
>>> lr.fit(X, y)
>>> lr.predict(X)
>>> lr.predict(X)
>>> est.score(X, y)
```

Methods

fit(X[, y])	Fit the model on the training data
get_params([deep])	Get parameters for this estimator.

Continued on next page

Table 18 – continued from previous page

predict(X)	Predict values for samples in X.
score(X, y)	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

__init__ (*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

family
The family this estimator is for.

fit (*X*, *y=None*)
Fit the model on the training data

Parameters

X: array-like, shape (n_samples, n_features)

y [array-like, shape (n_samples,)]

Returns

self [objectj]

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*X*)
Predict values for samples in X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples,]] Predicted value for each sample

score (*X*, *y*)
Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

dask_ml.linear_model.LogisticRegression

class dask_ml.linear_model.**LogisticRegression**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)

Esimator for logistic_regression.

Parameters

penalty [str or Regularizer, default 'l2'] Regularizer to use. Only relevant for the 'admm', 'lbfgs' and 'proximal_grad' solvers.

For string values, only 'l1' or 'l2' are valid.

dual [bool] Ignored

tol [float, default 1e-4] The tolerance for convergence.

C [float] Regularization strength. Note that `dask-glm` solvers use the parameterization $\lambda = 1/C$

fit_intercept [bool, default True] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

intercept_scaling [bool] Ignored

class_weight [dict or 'balanced'] Ignored

random_state [int, RandomState, or None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when solver == 'sag' or 'liblinear'.

solver [{'admm', 'gradient_descent', 'newton', 'lbfgs', 'proximal_grad'}] Solver to use. See Algorithms for details

multiclass [str, default 'ovr'] Ignored. Multiclass solvers not currently supported.

verbose [int, default 0] Ignored

warm_start [bool, default False] Ignored

n_jobs [int, default 1] Ignored

solver_kwargs [dict, optional, default None] Extra keyword arguments to pass through to the solver.

Attributes

coef_ [array, shape (n_classes, n_features)] The learned value for the model's coefficients

intercept_ [float of None] The learned value for the intercept, if one was added to the model

Examples

```
>>> from dask_glm.datasets import make_classification
>>> X, y = make_classification()
>>> lr = LogisticRegression()
>>> lr.fit(X, y)
>>> lr.predict(X)
>>> lr.predict_proba(X)
>>> est.score(X, y)
```

Methods

fit(X[, y])	Fit the model on the training data
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict class labels for samples in X.
predict_proba(X)	Probability estimates for samples in X.
score(X, y)	The mean accuracy on the given data and labels
set_params(**params)	Set the parameters of this estimator.

__init__(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

family
The family this estimator is for.

fit(*X*, *y=None*)
Fit the model on the training data

Parameters

X: array-like, shape (n_samples, n_features)

y [array-like, shape (n_samples,)]

Returns

self [objectj]

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)
Predict class labels for samples in X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples,]] Predicted class labels for each sample

predict_proba(*X*)
Probability estimates for samples in X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]] The probability of the sample for each class in the model.

score(*X*, *y*)
The mean accuracy on the given data and labels

Parameters

X [array-like, shape = [n_samples, n_features]] Test samples.

y [array-like, shape = [n_samples,]] Test labels.

Returns

score [float] Mean accuracy score

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

dask_ml.linear_model.PoissonRegression

class dask_ml.linear_model.**PoissonRegression**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)
Esimator for poisson_regression.

Parameters

penalty [str or Regularizer, default 'l2'] Regularizer to use. Only relevant for the 'admm', 'lbfgs' and 'proximal_grad' solvers.

For string values, only 'l1' or 'l2' are valid.

dual [bool] Ignored

tol [float, default 1e-4] The tolerance for convergence.

C [float] Regularization strength. Note that `dask-glm` solvers use the parameterization $\lambda = 1/C$

fit_intercept [bool, default True] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

intercept_scaling [bool] Ignored

class_weight [dict or 'balanced'] Ignored

random_state [int, RandomState, or None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when solver == 'sag' or 'liblinear'.

solver [{'admm', 'gradient_descent', 'newton', 'lbfgs', 'proximal_grad'}] Solver to use. See Algorithms for details

multiclass [str, default 'ovr'] Ignored. Multiclass solvers not currently supported.

verbose [int, default 0] Ignored

warm_start [bool, default False] Ignored

n_jobs [int, default 1] Ignored

solver_kwargs [dict, optional, default None] Extra keyword arguments to pass through to the solver.

Attributes

coef_ [array, shape (n_classes, n_features)] The learned value for the model's coefficients

intercept_ [float of None] The learned value for the intercept, if one was added to the model

Examples

```
>>> from dask_glm.datasets import make_counts
>>> X, y = make_counts()
>>> lr = PoissonRegression()
>>> lr.fit(X, y)
>>> lr.predict(X)
>>> lr.predict(X)
>>> lr.get_deviance(X, y)
```

Methods

fit(X[, y])	Fit the model on the training data

Continued on next page

Table 20 – continued from previous page

get_params([deep])	Get parameters for this estimator.
predict(X)	Predict count for samples in X.
set_params(**params)	Set the parameters of this estimator.

get_deviance

__init__ (*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1.0*, *class_weight=None*, *random_state=None*, *solver='admm'*, *multiclass='ovr'*, *verbose=0*, *warm_start=False*, *n_jobs=1*, *max_iter=100*, *solver_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

family
The family this estimator is for.

fit (*X*, *y=None*)
Fit the model on the training data

Parameters

X: array-like, shape (n_samples, n_features)

y [array-like, shape (n_samples,)]

Returns

self [objectj]

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*X*)
Predict count for samples in X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples,]] Predicted count for each sample

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

3.13.3 Meta-estimators for scikit-learn

dask-ml provides some meta-estimators that help use regular scikit-learn compatible estimators with Dask arrays.

wrappers.ParallelPostFit([estimator, scoring])	Meta-estimator for parallel predict and transform.
wrappers.Incremental([estimator, scoring, ...])	Metaestimator for feeding Dask Arrays to an estimator blockwise.

dask_ml.wrappers.ParallelPostFit

class dask_ml.wrappers.**ParallelPostFit**(*estimator=None*, *scoring=None*)

Meta-estimator for parallel predict and transform.

Parameters

estimator [Estimator] The underlying estimator that is fit.

scoring [string or callable, optional] A single string (see The scoring parameter: defining model evaluation rules) or a callable (see Defining your scoring strategy from metric functions) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See Specifying multiple metrics for evaluation for an example.

> **Warning:** If None, the estimator's default scorer (if available) is used. Most scikit-learn estimators will convert large Dask arrays to a single NumPy array, which may exhaust the memory of your worker. You probably want to always specify *scoring*.

See also:

Incremental

Notes

> **Warning:** This class is not appropriate for parallel or distributed *training* on large datasets. For that, see *Incremental*, which provides distributed (but sequential) training.

This estimator does not parallelize the training step. This simply calls the underlying estimators's `fit` method called and copies over the learned attributes to `self` afterwards.

It is helpful for situations where your training dataset is relatively small (fits on a single machine) but you need to predict or transform a much larger dataset. `predict`, `predict_proba` and `transform` will be done in parallel (potentially distributed if you've connected to a `dask.distributed.Client`).

Note that many scikit-learn estimators already predict and transform in parallel. This meta-estimator may still be useful in those cases when your dataset is larger than memory, as the distributed scheduler will ensure the data isn't all read into memory at once.

Examples

```
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> import sklearn.datasets
>>> import dask_ml.datasets
```

Make a small 1,000 sample 2 training dataset and fit normally.

```
>>> X, y = sklearn.datasets.make_classification(n_samples=1000,
...                                             random_state=0)
>>> clf = ParallelPostFit(estimator=GradientBoostingClassifier(),
...                       scoring='accuracy')
>>> clf.fit(X, y)
ParallelPostFit(estimator=GradientBoostingClassifier(...))
```

```
>>> clf.classes_
array([0, 1])
```

Transform and predict return dask outputs for dask inputs.

```
>>> X_big, y_big = dask_ml.datasets.make_classification(n_samples=100000,
                                                        random_state=0)
```

```
>>> clf.predict(X)
dask.array<predict, shape=(10000,), dtype=int64, chunksize=(1000,)>
```

Which can be computed in parallel.

```
>>> clf.predict_proba(X).compute()
array([[0.99141094, 0.00858906],
       [0.93178389, 0.06821611],
       [0.99129105, 0.00870895],
       ...,
       [0.97996652, 0.02003348],
       [0.98087444, 0.01912556],
       [0.99407016, 0.00592984]])
```

Methods

fit(X[, y])	Fit the underlying estimator.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict for X.
predict_proba(X)	Predict for X.
score(X, y)	Returns the score on the given data.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform block or partition-wise for dask inputs.

__init__ (*estimator=None*, *scoring=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y=None*, ***kwargs*)
Fit the underlying estimator.

Parameters

X, y [array-like]

****kwargs** Additional fit-kwargs for the underlying estimator.

Returns

self [object]

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*X*)
Predict for X.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

Parameters

X [array-like]

Returns

y [array-like]

predict_proba (*X*)
Predict for X.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

If the underlying estimator does not have a `predict_proba` method, then an `AttributeError` is raised.

Parameters

X [array or dataframe]

Returns

y [array-like]

score (*X*, *y*)
Returns the score on the given data.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float] return self.estimator.score(X, y)

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)
Transform block or partition-wise for dask inputs.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

If the underlying estimator does not have a `transform` method, then an `AttributeError` is raised.

Parameters

X [array-like]

Returns

transformed [array-like]

dask_ml.wrappers.Incremental

class dask_ml.wrappers.**Incremental**(*estimator=None*, *scoring=None*, *shuffle_blocks=True*, *random_state=None*)
Metaestimator for feeding Dask Arrays to an estimator blockwise.

This wrapper provides a bridge between Dask objects and estimators implementing the `partial_fit` API. These *incremental learners* can train on batches of data. This fits well with Dask's blocked data structures.

See the list of incremental learners in the scikit-learn documentation for a list of estimators that implement the `partial_fit` API. Note that *Incremental* is not limited to just these classes, it will work on any estimator implementing `partial_fit`, including those defined outside of scikit-learn itself.

Calling *Incremental.fit()* with a Dask Array will pass each block of the Dask array or arrays to `estimator.partial_fit` *sequentially*.

Like *ParallelPostFit*, the methods available after fitting (e.g. *Incremental.predict()*, etc.) are all parallel and delayed.

The `estimator_` attribute is a clone of *estimator* that was actually used during the call to `fit`. All attributes learned during training are available on `Incremental` directly.

Parameters

estimator [Estimator] Any object supporting the scikit-learn `parital_fit` API.

scoring [string or callable, optional] A single string (see The scoring parameter: defining model evaluation rules) or a callable (see Defining your scoring strategy from metric functions) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See Specifying multiple metrics for evaluation for an example.

> **Warning:** If None, the estimator's default scorer (if available) is used. Most scikit-learn estimators will convert large Dask arrays to a single NumPy array, which may exhaust the memory of your worker. You probably want to always specify *scoring*.

random_state [int or numpy.random.RandomState, optional] Random object that determines how to shuffle blocks.

shuffle_blocks [bool, default True] Determines whether to call `partial_fit` on a randomly selected chunk of the Dask arrays (default), or to fit in sequential order. This does not control shuffle between blocks or shuffling each block.

Attributes

estimator_ [Estimator] A clone of *estimator* that was actually fit during the `.fit` call.

See also:

ParallelPostFit

Examples

```
>>> from dask_ml.wrappers import Incremental
>>> from dask_ml.datasets import make_classification
>>> import sklearn.linear_model
>>> X, y = make_classification(chunks=25)
>>> est = sklearn.linear_model.SGDClassifier()
>>> clf = Incremental(est, scoring='accuracy')
>>> clf.fit(X, y, classes=[0, 1])
```

When used inside a grid search, prefix the underlying estimator's parameter names with `estimator__`.

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {"estimator__alpha": [0.1, 1.0, 10.0]}
>>> gs = GridSearchCV(clf, param_grid)
>>> gs.fit(X, y, classes=[0, 1])
```

Methods

fit(X[, y])	Fit the underlying estimator.
get_params([deep])	Get parameters for this estimator.
partial_fit(X[, y])	Fit the underlying estimator.
predict(X)	Predict for X.
predict_proba(X)	Predict for X.

Continued on next page

Table 23 – continued from previous page

score(X, y)	Returns the score on the given data.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform block or partition-wise for dask inputs.

__init__ (*estimator=None*, *scoring=None*, *shuffle_blocks=True*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y=None*, ***fit_kwargs*)
Fit the underlying estimator.

Parameters

X, y [array-like]

****kwargs** Additional fit-kwargs for the underlying estimator.

Returns

self [object]

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

partial_fit (*X*, *y=None*, ***fit_kwargs*)
Fit the underlying estimator.

If this estimator has not been previously fit, this is identical to *Incremental.fit()*. If it has been previously fit, `self.estimator_` is used as the starting point.

Parameters

X, y [array-like]

****kwargs** Additional fit-kwargs for the underlying estimator.

Returns

self [object]

predict (*X*)
Predict for X.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

Parameters

X [array-like]

Returns

y [array-like]

predict_proba (*X*)
Predict for X.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

If the underlying estimator does not have a `predict_proba` method, then an `AttributeError` is raised.

Parameters

X [array or dataframe]

Returns

y [array-like]

score(*X*, *y*)

Returns the score on the given data.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float] return self.estimator.score(X, y)

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)

Transform block or partition-wise for dask inputs.

For dask inputs, a dask array or dataframe is returned. For other inputs (NumPy array, pandas dataframe, scipy sparse matrix), the regular return value is returned.

If the underlying estimator does not have a `transform` method, then an `AttributeError` is raised.

Parameters

X [array-like]

Returns

transformed [array-like]

3.13.4 dask_ml.cluster: Clustering

Unsupervised Clustering Algorithms

cluster.KMeans([n_clusters, init, ...])	Scalable KMeans for clustering
cluster.SpectralClustering([n_clusters, ...])	Apply parallel Spectral Clustering

dask_ml.cluster.KMeans

class dask_ml.cluster.**KMeans**(*n_clusters=8*, *init='k-means||'*, *oversampling_factor=2*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *random_state=None*, *copy_x=True*, *n_jobs=1*, *algorithm='full'*, *init_max_iter=None*)

Scalable KMeans for clustering

Parameters

n_clusters [int, default 8] Number of clusters to end up with

init [{'k-means||', 'k-means++' or ndarray}] Method for center initialization, defualts to 'k-means||'.

'k-means||' : selects the the gg

'k-means++' : selects the initial cluster centers in a smart way to speed up convergence. Uses scikit-learn's implementation.

> **Warning:** If using `'k-means++'`, the entire dataset will be read into memory at once.

An array of shape (n_clusters, n_features) can be used to give an explicit starting point

oversampling_factor [int, default 2] Oversampling factor for use in the `k-means||` algorithm.

max_iter [int] Maximum number EM iterations to attempt.

init_max_iter [int] Number of iterations for init step.

tol [float] Relative tolerance with regards to inertia to declare convergence

algorithm ['full'] The algorithm to use for the EM step. Only "full" (LLoyd's algorithm) is allowed.

random_state [int, RandomState instance or None, optional, default: None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

cluster_centers_ [np.ndarray [n_clusters, n_features]] A NumPy array with the cluster centers

labels_ [da.array [n_samples,]] A dask array with the index position in `cluster_centers_` this sample belongs to.

intertia_ [float] Sum of distances of samples to their closest cluster center.

n_iter_ [int] Number of EM steps to reach convergence

See also:

`PartialMiniBatchKMeans`, `sklearn.cluster.MiniBatchKMeans`, `sklearn.cluster.KMeans`

Notes

This class implements a parallel and distributed version of k-Means.

Initialization with k-means||

The default initializer for `KMeans` is `k-means||`, compared to `k-means++` from scikit-learn. This is the algorithm described in *Scalable K-Means++ (2012)*.

`k-means||` is designed to work well in a distributed environment. It's a variant of k-means++ that's designed to work in parallel (k-means++ is inherently sequential). Currently, the `k-means||` implementation here is slower than scikit-learn's `k-means++` if your entire dataset fits in memory on a single machine. If that's the case, consider using `init='k-means++'`.

Parallel Lloyd's Algorithm

LLoyd's Algorithm (the default Expectation Maximization algorithm used in scikit-learn) is naturally parallelizable. In naive benchmarks, the implementation here achieves 2-3x speedups over scikit-learn.

Both the initialization step and the EM steps make multiple passes over the data. If possible, persist your dask collections in (distributed) memory before running `.fit`.

References

- Scalable K-Means++, 2012 Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, Sergei Vassilvitskii https://arxiv.org/abs/1203.6402

Methods

fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.

fit	
transform	

__init__ (*n_clusters=8*, *init='k-means||'*, *oversampling_factor=2*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *random_state=None*, *copy_x=True*, *n_jobs=1*, *algorithm='full'*, *init_max_iter=None*)

Initialize self. See help(type(self)) for accurate signature.

fit_transform (*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict(*X*)

Predict the closest cluster each sample in X belongs to. In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters

X [array-like, shape = [n_samples, n_features]] New data to predict.

Returns

labels [array, shape [n_samples,]] Index of the cluster each sample belongs to.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

dask_ml.cluster.SpectralClustering

class dask_ml.cluster.**SpectralClustering**(*n_clusters=8*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=1*, *n_components=100*, *persist_embedding=False*, *kmeans_params=None*)

Apply parallel Spectral Clustering

This implementation avoids the expensive computation of the N x N affinity matrix. Instead, the Nystrom Method is used as an approximation.

Parameters

n_clusters [integer, optional] The dimension of the projection subspace.

eigen_solver [None] ignored

random_state [int, RandomState instance or None, optional, default: None] A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg' and by the K-Means initialization. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

n_init [int, optional, default: 10] ignored

gamma [float, default=1.0] Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. Ignored for `affinity='nearest_neighbors'`.

affinity [string, array-like or callable, default 'rbf'] If a string, this may be one of 'nearest_neighbors', 'precomputed', 'rbf' or one of the kernels supported by *sklearn.metrics.pairwise_kernels*.

Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.

Callables should expect arguments similar to *sklearn.metrics.pairwise_kernels*: a required `X`, an optional `Y`, and `gamma`, `degree`, `coef0`, and any keywords passed in `kernel_params`.

n_neighbors [integer] Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity='rbf'`.

eigen_tol [float, optional, default: 0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

assign_labels ['kmeans' or Estimator, default: 'kmeans'] The strategy to use to assign labels in the embedding space. By default creates an instance of *dask_ml.cluster.KMeans* and sets *n_clusters* to 2. For further control over the hyperparameters of the final label assignment, pass an instance of a `KMeans` estimator (either scikit-learn or dask-ml).

degree [float, default=3] Degree of the polynomial kernel. Ignored by other kernels.

coef0 [float, default=1] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params [dictionary of string to any, optional] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

n_jobs [int, optional (default = 1)] The number of parallel jobs to run. If `-1`, then the number of jobs is set to the number of CPU cores.

n_components [int, default 100] Number of rows from `X` to use for the Nystrom approximation. Larger `n_components` will improve the accuracy of the approximation, at the cost of a longer training time.

persist_embedding [bool] Whether to persist the intermediate n_samples x n_components array used for clustering.

kmeans_params [dictionary of string to any, optional] Keyword arguments for the KMeans clustering used for the final clustering.

Attributes

assign_labels_ [Estimator] The instance of the KMeans estimator used to assign labels

labels_ [dask.array.Array, size (n_samples,)] The cluster labels assigned

eigenvalues_ [numpy.ndarray] The eigenvalues from the SVD of the sampled points

Notes

Using `persist_embedding=True` can be an important optimization to avoid some redundant computations. This persists the array being fed to the clustering algorithm in (distributed) memory. The array is shape `n_samples x n_components`.

References

- Parallel Spectral Clustering in Distributed Systems, 2010 Chen, Song, Bai, Lin, and Chang IEEE Transactions on Pattern Analysis and Machine Intelligence http://ieeexplore.ieee.org/document/5444877/
- Spectral Grouping Using the Nystrom Method (2004) Fowlkes, Belongie, Chung, Malik IEEE Transactions on Pattern Analysis and Machine Intelligence https://people.cs.umass.edu/~mahadeva/cs791bb/reading/fowlkes-nystrom.pdf

Methods

fit_predict(X[, y])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit

__init__ (*n_clusters=8*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=1*, *n_components=100*, *persist_embedding=False*, *kmeans_params=None*)
Initialize self. See help(type(self)) for accurate signature.

fit_predict (*X*, *y=None*)
Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

Returns

y [ndarray, shape (n_samples,)] cluster labels

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

3.13.5 dask_ml.decomposition: Matrix Decomposition

decomposition.PCA([n_components, copy, ...])	Principal component analysis (PCA)
decomposition.TruncatedSVD([n_components, ...])	

dask_ml.decomposition.PCA

class dask_ml.decomposition.**PCA**(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power=0*, *random_state=None*)

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the "tsqr" algorithm from Benson et. al. (2013). See the References for more.

Read more in the User Guide.

Parameters

n_components [int, or None] Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

Note: Unlike scikit-learn, `n_components='mle'` and `n_components` between `(0, 1)` are not currently supported.

copy [bool (default True)] ignored

whiten [bool, optional (default False)] When True (False by default) the *components_* vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

svd_solver [string {'auto', 'full', 'tsqr', 'randomized'}]

auto : the solver is selected by a default policy based on *X.shape* and *n_components*: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

full : run exact full SVD and select the components by postprocessing

randomized : run randomized SVD by using `da.linalg.svd_compressed`.

tol [float >= 0, optional (default .0)] ignored

iterated_power [int >= 0, default 0] Number of iterations for the power method computed by svd_solver == 'randomized'.

random_state [int, RandomState instance or None, optional (default None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *da.random*. Used when `svd_solver` == 'randomized'.

Attributes

components_ [array, shape (n_components, n_features)] Principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by `explained_variance_`.

explained_variance_ [array, shape (n_components,)] The amount of variance explained by each of the selected components.

Equal to n_components largest eigenvalues of the covariance matrix of X.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

If `n_components` is not set then all components are stored and the sum of the ratios is equal to 1.0.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set.

Equal to *X.mean(axis=0)*.

n_components_ [int] The estimated number of components. When n_components is set to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this number is estimated from input data. Otherwise it equals the parameter n_components, or the lesser value of n_features and n_samples if n_components is None.

noise_variance_ [float] The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999. See "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf. It is required to computed the estimated data covariance and score samples.

Equal to the average of (min(n_features, n_samples) - n_components) smallest eigenvalues of the covariance matrix of X.

Notes

Differences from scikit-learn:

- svd_solver : 'randomized' uses `dask.linalg.svd_compressed` 'full' uses `dask.linalg.svd`, 'arpack' is not valid.
- iterated_power : defaults to `0`, the default for `dask.linalg.svd_compressed`.
- n_components : `n_components='mle'` is not allowed. Fractional `n_components` between 0 and 1 is not allowed.

References

Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. A. Benson, D. Gleich, and J. Demmel. IEEE International Conference on Big Data, 2013. http://arxiv.org/abs/1301.1071

Examples

```
>>> import numpy as np
>>> import dask.array as da
>>> from dask_ml.decomposition import PCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> dX = da.from_array(X, chunks=X.shape)
>>> pca = PCA(n_components=2)
>>> pca.fit(dX)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[ 0.99244...  0.00755...]
>>> print(pca.singular_values_)
[ 6.30061...  0.54980...]
```

```
>>> pca = PCA(n_components=2, svd_solver='full')
>>> pca.fit(dX)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='full', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[ 0.99244...  0.00755...]
>>> print(pca.singular_values_)
[ 6.30061...  0.54980...]
```

Methods

fit(X[, y])	Placeholder for fit.
fit_transform(X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_covariance()	Compute data covariance with the generative model.
get_params([deep])	Get parameters for this estimator.
get_precision()	Compute data precision matrix with the generative model.
inverse_transform(X)	Transform data back to its original space.
score(X[, y])	Return the average log-likelihood of all samples.
score_samples(X)	Return the log-likelihood of each sample.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Apply dimensionality reduction on X.

__init__(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y=None*)
Placeholder for fit. Subclasses should implement this method!

Fit the model with X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data, where n_samples is the number of samples and n_features is the number of features.

Returns

self [object] Returns the instance itself.

fit_transform(*X*, *y=None*)
Fit the model with X and apply the dimensionality reduction on X.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

X_new [array-like, shape (n_samples, n_components)]

get_covariance()
Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)`
where S**2 contains the explained variances, and sigma2 contains the noise variances.

Returns

cov [array, shape=(n_features, n_features)] Estimated covariance of data.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

get_precision()
Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returns

precision [array, shape=(n_features, n_features)] Estimated precision of data.

inverse_transform(*X*)
Transform data back to its original space.

Returns an array X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data, where n_samples in the number of samples and n_components is the number of components.

Returns

X_original array-like, shape (n_samples, n_features)

Notes

If whitening is enabled, inverse_transform does not compute the exact inverse operation of transform.

score(*X*, *y=None*)
Return the average log-likelihood of all samples.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

y [Ignored]

Returns

ll [float] Average log-likelihood of the samples under the current model

score_samples(*X*)
Return the log-likelihood of each sample.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

Returns

ll [array, shape (n_samples,)] Log-likelihood of each sample under the current model

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)
Apply dimensionality reduction on X.

X is projected on the first principal components previous extracted from a training set.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape (n_samples, n_components)]

dask_ml.decomposition.TruncatedSVD

class dask_ml.decomposition.**TruncatedSVD**(*n_components=2*, *algorithm='tsqr'*, *n_iter=5*, *random_state=None*, *tol=0.0*)

Methods

fit(X[, y])	Fit truncated SVD on training data X
fit_transform(X[, y])	Fit model to X and perform dimensionality reduction on X.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Transform X back to its original space.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Perform dimensionality reduction on X.

__init__ (*n_components=2*, *algorithm='tsqr'*, *n_iter=5*, *random_state=None*, *tol=0.0*)

Dimensionality reduction using truncated SVD (aka LSA).

This transformer performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). Contrary to PCA, this estimator does not center the data before computing the singular value decomposition.

Parameters

n_components [int, default = 2] Desired dimensionality of output data. Must be less than or equal to the number of features. The default value is useful for visualization.

algorithm [{'tsqr', 'randomized'}] SVD solver to use. Both use the *tsqr* (for "tall-and-skinny QR") algorithm internally. 'randomized' uses an approximate algorithm that is faster, but not exact. See the References for more.

n_iter [int, optional (default 0)] Number of power iterations, useful when the singular values decay slowly. Error decreases exponentially as n_power_iter increases. In practice, set n_power_iter <= 4.

random_state [int, RandomState instance or None, optional] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

tol [float, optional] Ignored.

Attributes

components_ [array, shape (n_components, n_features)]

explained_variance_ [array, shape (n_components,)] The variance of the training samples transformed by a projection to each component.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

See also:

`dask.array.linalg.tsqr`, `dask.array.linalg.svd_compressed`

Notes

SVD suffers from a problem called "sign indeterminacy", which means the sign of the `components_` and the output from transform depend on the algorithm and random state. To work around this, fit instances of this class to data once, then keep the instance around to do transformations.

Warning: The implementation currently does not support sparse matricies.

References

Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. A. Benson, D. Gleich, and J. Demmel. IEEE International Conference on Big Data, 2013. http://arxiv.org/abs/1301.1071

Examples

```
>>> from dask_ml.decomposition import TruncatedSVD
>>> import dask.array as da
>>> X = da.random.normal(size=(1000, 20), chunks=(100, 20))
>>> svd = TruncatedSVD(n_components=5, n_iter=3, random_state=42)
>>> svd.fit(X)
TruncatedSVD(algorithm='tsqr', n_components=5, n_iter=3,
             random_state=42, tol=0.0)
```

```
>>> print(svd.explained_variance_ratio_)
[0.06386323 0.06176776 0.05901293 0.0576399  0.05726607]
>>> print(svd.explained_variance_ratio_.sum())
0.299...
>>> print(svd.singular_values_)
array([35.92469517, 35.32922121, 34.53368856, 34.138..., 34.013...])
```

Note that `tranform` returns a `dask.Array`.

```
>>> svd.transform(X)
dask.array<sum-agg, shape=(1000, 5), dtype=float64, chunksize=(100, 5)>
```

fit (*X*, *y=None*)
Fit truncated SVD on training data X

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

self [object] Returns the transformer object.

fit_transform (*X*, *y=None*)
Fit model to X and perform dimensionality reduction on X.

Parameters

X [array-like, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array, of the same type as the input array. If `X` was a `dask.array`, then `X_new` will be a `dask.array` with the same chunks along the first dimension.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Transform X back to its original space.

Returns an array X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data.

Returns

X_original [array, shape (n_samples, n_features)] Note that this is always a dense array.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*)
Perform dimensionality reduction on X.

Parameters

X [array-like, shape (n_samples, n_features)] Data to be transformed.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array, of the same type as the input array. If `X` was a `dask.array`, then `X_new` will be a `dask.array` with the same chunks along the first dimension.

3.13.6 `dask_ml.preprocessing`: Preprocessing Data

Utilties for Preprocessing data.

preprocessing.StandardScaler([copy, ...])	Standardize features by removing the mean and scaling to unit variance
preprocessing.RobustScaler([with_centering, ...])	Scale features using statistics that are robust to outliers.
preprocessing.MinMaxScaler([feature_range, copy])	Transforms features by scaling each feature to a given range.
preprocessing.QuantileTransformer([...])	Transforms features using quantile information.

Continued on next page

Table 30 – continued from previous page

preprocessing.Categorizer([categories, columns])	Transform columns of a DataFrame to categorical dtype.
preprocessing.DummyEncoder([columns, drop_first])	Dummy (one-hot) encode categorical columns.
preprocessing.OrdinalEncoder([columns])	Ordinal (integer) encode categorical columns.
preprocessing.LabelEncoder	Encode labels with value between 0 and n_classes-1.

dask_ml.preprocessing.StandardScaler

class dask_ml.preprocessing.**StandardScaler**(*copy=True*, *with_mean=True*, *with_std=True*)

Standardize features by removing the mean and scaling to unit variance

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the *transform* method.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

This scaler can also be applied to sparse CSR or CSC matrices by passing *with_mean=False* to avoid breaking the sparsity structure of the data.

Read more in the User Guide.

Parameters

copy [boolean, optional, default True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

with_mean [boolean, True by default] If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

Attributes

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data.

New in version 0.17: *scale_*

mean_ [array of floats with shape [n_features]] The mean value for each feature in the training set.

var_ [array of floats with shape [n_features]] The variance for each feature in the training set. Used to compute *scale_*

n_samples_seen_ [int] The number of samples processed by the estimator. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

scale Equivalent function without the estimator API.

sklearn.decomposition.PCA Further removes the linear correlation across features with 'whiten=True'.

Notes

For a comparison of the different scalers, transformers, and normalizers, see examples/preprocessing/plot_all_scaling.py.

Examples

```
>>> from sklearn.preprocessing import StandardScaler
>>>
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> print(scaler.mean_)
[ 0.5  0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[ 3.  3.]]
```

Methods

fit(X[, y])	Compute the mean and std to be used for later scaling.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X[, copy])	Scale back the data to the original representation
partial_fit(X[, y])	Online computation of mean and std on X for later scaling.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y, copy])	Perform standardization by centering and scaling

__init__ (*copy=True*, *with_mean=True*, *with_std=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y=None*)
Compute the mean and std to be used for later scaling.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [Passthrough for `Pipeline` compatibility.]

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*, *copy=None*)
Scale back the data to the original representation

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

copy [bool, optional (default: None)] Copy the input X or not.

Returns

X_tr [array-like, shape [n_samples, n_features]] Transformed array.

partial_fit(*X*, *y=None*)
Online computation of mean and std on X for later scaling. All of X is processed as a single batch. This is intended for cases when *fit* is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

The algorithm for incremental mean and std is given in Equation 1.5a,b in Chan, Tony F., Gene H. Golub, and Randall J. LeVeque. "Algorithms for computing the sample variance: Analysis and recommendations." The American Statistician 37.3 (1983): 242-247:

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [Passthrough for `Pipeline` compatibility.]

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*, *copy=None*)

Perform standardization by centering and scaling

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

y [(ignored)] Deprecated since version 0.19: This parameter will be removed in 0.21.

copy [bool, optional (default: None)] Copy the input X or not.

dask_ml.preprocessing.RobustScaler

class dask_ml.preprocessing.**RobustScaler**(*with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

Scale features using statistics that are robust to outliers.

This Scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling happen independently on each feature (or each sample, depending on the `axis` argument) by computing the relevant statistics on the samples in the training set. Median and interquartile range are then stored to be used on later data using the `transform` method.

Standardization of a dataset is a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean / variance in a negative way. In such cases, the median and the interquartile range often give better results.

New in version 0.17.

Read more in the User Guide.

Parameters

with_centering [boolean, True by default] If True, center the data before scaling. This will cause `transform` to raise an exception when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_scaling [boolean, True by default] If True, scale the data to interquartile range.

quantile_range [tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0] Default: (25.0, 75.0) = (1st quantile, 3rd quantile) = IQR Quantile range used to calculate `scale_`.

New in version 0.18.

copy [boolean, optional, default is True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

Attributes

center_ [array of floats] The median value for each feature in the training set.

scale_ [array of floats] The (scaled) interquartile range for each feature in the training set.

New in version 0.17: *scale_* attribute.

See also:

robust_scale Equivalent function without the estimator API.

sklearn.decomposition.PCA Further removes the linear correlation across features with 'whiten=True'.

Notes

For a comparison of the different scalers, transformers, and normalizers, see examples/preprocessing/plot_all_scaling.py.

https://en.wikipedia.org/wiki/Median_(statistics) https://en.wikipedia.org/wiki/Interquartile_range

Methods

fit(X[, y])	Compute the median and quantiles to be used for scaling.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Scale back the data to the original representation
set_params(**params)	Set the parameters of this estimator.
transform(X)	Center and scale the data.

__init__(*with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y=None*)
Compute the median and quantiles to be used for scaling.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the median and quantiles used for later scaling along the features axis.

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Scale back the data to the original representation

Parameters

X [array-like] The data used to scale along the specified axis.

This implementation was copied and modified from Scikit-Learn.

See License information here:

https://github.com/scikit-learn/scikit-learn/blob/master/README.rst

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)
Center and scale the data.

Can be called on sparse input, provided that `RobustScaler` has been fitted to dense input and `with_centering=False`.

Parameters

X [{array-like, sparse matrix}] The data used to scale along the specified axis.

This implementation was copied and modified from Scikit-Learn.

See License information here:

https://github.com/scikit-learn/scikit-learn/blob/master/README.rst

dask_ml.preprocessing.MinMaxScaler

class dask_ml.preprocessing.**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)
Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, i.e. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the User Guide.

Parameters

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

copy [boolean, optional, default True] Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

Attributes

min_ [ndarray, shape (n_features,)] Per feature adjustment for minimum.

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data.

New in version 0.17: *scale_* attribute.

data_min_ [ndarray, shape (n_features,)] Per feature minimum seen in the data

New in version 0.17: *data_min_*

data_max_ [ndarray, shape (n_features,)] Per feature maximum seen in the data

New in version 0.17: *data_max_*

data_range_ [ndarray, shape (n_features,)] Per feature range `(data_max_ - data_min_)` seen in the data

New in version 0.17: *data_range_*

See also:

`minmax_scale` Equivalent function without the estimator API.

Notes

For a comparison of the different scalers, transformers, and normalizers, see examples/preprocessing/plot_all_scaling.py.

Examples

```
>>> from sklearn.preprocessing import MinMaxScaler
>>>
>>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>>> scaler = MinMaxScaler()
>>> print(scaler.fit(data))
MinMaxScaler(copy=True, feature_range=(0, 1))
>>> print(scaler.data_max_)
[  1.  18.]
>>> print(scaler.transform(data))
[[ 0.    0.  ]
 [ 0.25  0.25]
 [ 0.5   0.5 ]
 [ 1.    1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[ 1.5  0. ]]
```

Methods

fit(X[, y])	Compute the minimum and maximum to be used for later scaling.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.

Continued on next page

Table 33 – continued from previous page

inverse_transform(X[, y, copy])	Undo the scaling of X according to feature_range.
partial_fit(X[, y])	Online computation of min and max on X for later scaling.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y, copy])	Scaling features of X according to feature_range.

__init__ (*feature_range=(0, 1)*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y=None*)
Compute the minimum and maximum to be used for later scaling.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the per-feature minimum and maximum used for later scaling along the features axis.

fit_transform (*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform (*X*, *y=None*, *copy=None*)
Undo the scaling of X according to feature_range.

Parameters

X [array-like, shape [n_samples, n_features]] Input data that will be transformed. It cannot be sparse.

partial_fit (*X*, *y=None*)
Online computation of min and max on X for later scaling. All of X is processed as a single batch. This is intended for cases when *fit* is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y [Passthrough for `Pipeline` compatibility.]

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*, *copy=None*)
Scaling features of X according to feature_range.

Parameters

X [array-like, shape [n_samples, n_features]] Input data that will be transformed.

dask_ml.preprocessing.QuantileTransformer

class dask_ml.preprocessing.**QuantileTransformer**(*n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)
Transforms features using quantile information.

This implementation differs from the scikit-learn implementation by using approximate quantiles. The scikit-learn docstring follows.

This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

The transformation is applied on each feature independently. The cumulative density function of a feature is used to project the original values. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

Read more in the User Guide.

Parameters

n_quantiles [int, optional (default=1000)] Number of quantiles to be computed. It corresponds to the number of landmarks used to discretize the cumulative density function.

output_distribution [str, optional (default='uniform')] Marginal distribution for the transformed data. The choices are 'uniform' (default) or 'normal'.

ignore_implicit_zeros [bool, optional (default=False)] Only applies to sparse matrices. If True, the sparse entries of the matrix are discarded to compute the quantile statistics. If False, these entries are treated as zeros.

subsample [int, optional (default=1e5)] Maximum number of samples used to estimate the quantiles for computational efficiency. Note that the subsampling procedure may differ for value-identical sparse and dense matrices.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is

the RandomState instance used by np.random. Note that this is used by subsampling and smoothing noise.

copy [boolean, optional, (default=True)] Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array).

Attributes

quantiles_ [ndarray, shape (n_quantiles, n_features)] The values corresponding the quantiles of reference.

references_ [ndarray, shape(n_quantiles,)] Quantiles of references.

See also:

`quantile_transform` Equivalent function without the estimator API.

`StandardScaler` perform standardization that is faster, but less robust to outliers.

`RobustScaler` perform robust standardization that removes the influence of outliers but does not put outliers and inliers on the same scale.

Notes

For a comparison of the different scalers, transformers, and normalizers, see examples/preprocessing/plot_all_scaling.py.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import QuantileTransformer
>>> rng = np.random.RandomState(0)
>>> X = np.sort(rng.normal(loc=0.5, scale=0.25, size=(25, 1)), axis=0)
>>> qt = QuantileTransformer(n_quantiles=10, random_state=0)
>>> qt.fit_transform(X)
array([...])
```

Methods

fit(X[, y])	Compute the quantiles used for transforming.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Back-projection to the original space.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Feature-wise transformation of the data.

__init__(*n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y=None*)
Compute the quantiles used for transforming.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if *ignore_implicit_zeros* is False.

Returns

self [object] Returns self

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Back-projection to the original space.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if *ignore_implicit_zeros* is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*)
Feature-wise transformation of the data.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if *ignore_implicit_zeros* is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

dask_ml.preprocessing.Categorizer

class dask_ml.preprocessing.**Categorizer**(*categories=None*, *columns=None*)

Transform columns of a DataFrame to categorical dtype.

This is a useful pre-processing step for dummy, one-hot, or categorical encoding.

Parameters

categories [mapping, optional] A dictionary mapping column name to instances of `pandas.api.types.CategoricalDtype`. Alternatively, a mapping of column name to `(categories, ordered)` tuples.

columns [sequence, optional] A sequence of column names to limit the categorization to. This argument is ignored when `categories` is specified.

Attributes

columns_ [pandas.Index] The columns that were categorized. Useful when `categories` is None, and we detect the categorical and object columns

categories_ [dict] A dictionary mapping column names to dtypes. For pandas>=0.21.0, the values are instances of `pandas.api.types.CategoricalDtype`. For older pandas, the values are tuples of `(categories, ordered)`.

Notes

This transformer only applies to `dask.DataFrame` and `pandas.DataFrame`. By default, all object-type columns are converted to categoricals. The set of categories will be the values present in the column and the categoricals will be unordered. Pass `dtypes` to control this behavior.

All other columns are included in the transformed output untouched.

For `dask.DataFrame`, any unknown categoricals will become known.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": ['a', 'a', 'b']})
>>> ce = Categorizer()
>>> ce.fit_transform(df).dtypes
A       int64
B    category
dtype: object
```

```
>>> ce.categories_
{'B': CategoricalDtype(categories=['a', 'b'], ordered=False)}
```

Using CategoricalDtypes for specifying the categories:

```
>>> from pandas.api.types import CategoricalDtype
>>> ce = Categorizer(categories={"B": CategoricalDtype(['a', 'b', 'c'])})
>>> ce.fit_transform(df).B.dtype
CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)
```

Methods

fit(X[, y])	Find the categorical columns.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Transform the columns in `X` according to `self.categories_`.

__init__ (*categories=None*, *columns=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y=None*)
Find the categorical columns.

Parameters

X [pandas.DataFrame or dask.DataFrame]

y [ignored]

Returns

self

fit_transform (*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*)
Transform the columns in `X` according to `self.categories_`.

Parameters

X [pandas.DataFrame or dask.DataFrame]

y [ignored]

Returns

X_trn [pandas.DataFrame or dask.DataFrame] Same type as the input. The columns in `self.categories_` will be converted to categorical dtype.

dask_ml.preprocessing.DummyEncoder

class dask_ml.preprocessing.**DummyEncoder**(*columns=None*, *drop_first=False*)
Dummy (one-hot) encode categorical columns.

Parameters

columns [sequence, optional] The columns to dummy encode. Must be categorical dtype. Dummy encodes all categorical dtype columns by default.

drop_first [bool, default False] Whether to drop the first category in each column.

Attributes

columns_ [Index] The columns in the training data before dummy encoding

transformed_columns_ [Index] The columns in the training data after dummy encoding

categorical_columns_ [Index] The categorical columns in the training data

noncategorical_columns_ [Index] The rest of the columns in the training data

categorical_blocks_ [dict] Mapping from column names to slice objects. The slices represent the positions in the transformed array that the categorical column ends up at

dtypes_ [dict] Dictionary mapping column name to either

- instances of CategoricalDtype (pandas >= 0.21.0)
- tuples of (categories, ordered)

Notes

This transformer only applies to dask and pandas DataFrames. For dask DataFrames, all of your categoricals should be known.

The inverse transformation can be used on a dataframe or array.

Examples

```
>>> data = pd.DataFrame({"A": [1, 2, 3, 4],
...                      "B": pd.Categorical(['a', 'a', 'a', 'b'])})
>>> de = DummyEncoder()
>>> trn = de.fit_transform(data)
>>> trn
A  B_a  B_b
0  1    1    0
1  2    1    0
2  3    1    0
3  4    0    1
```

```
>>> de.columns_
Index(['A', 'B'], dtype='object')
```

```
>>> de.non_categorical_columns_
Index(['A'], dtype='object')
```

```
>>> de.categorical_columns_
Index(['B'], dtype='object')
```

```
>>> de.dtypes_
{'B': CategoricalDtype(categories=['a', 'b'], ordered=False)}
```

```
>>> de.categorical_blocks_
{'B': slice(1, 3, None)}
```

```
>>> de.fit_transform(dd.from_pandas(data, 2))
Dask DataFrame Structure:
                A    B_a    B_b
npartitions=2
0              int64  uint8  uint8
2                ...    ...    ...
3                ...    ...    ...
Dask Name: get_dummies, 4 tasks
```

Methods

fit(X[, y])	Determine the categorical columns to be dummy encoded.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Inverse dummy-encode the columns in *X*
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Dummy encode the categorical columns in X

__init__(*columns=None*, *drop_first=False*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y=None*)

Determine the categorical columns to be dummy encoded.

Parameters

X [pandas.DataFrame or dask.dataframe.DataFrame]

y [ignored]

Returns

self

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Inverse dummy-encode the columns in *X*

Parameters

X [array or dataframe] Either the NumPy, dask, or pandas version

Returns

data [DataFrame] Dask array or dataframe will return a Dask DataFrame. Numpy array or pandas dataframe will return a pandas DataFrame

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*)
Dummy encode the categorical columns in X

Parameters

X [pd.DataFrame or dd.DataFrame]

y [ignored]

Returns

transformed [pd.DataFrame or dd.DataFrame] Same type as the input

dask_ml.preprocessing.OrdinalEncoder

class dask_ml.preprocessing.**OrdinalEncoder**(*columns=None*)

Ordinal (integer) encode categorical columns.

Parameters

columns [sequence, optional] The columns to encode. Must be categorical dtype. Encodes all categorical dtype columns by default.

Attributes

columns_ [Index] The columns in the training data before/after encoding

categorical_columns_ [Index] The categorical columns in the training data

noncategorical_columns_ [Index] The rest of the columns in the training data

dtypes_ [dict] Dictionary mapping column name to either

- instances of CategoricalDtype (pandas >= 0.21.0)
- tuples of (categories, ordered)

Notes

This transformer only applies to dask and pandas DataFrames. For dask DataFrames, all of your categoricals should be known.

The inverse transformation can be used on a dataframe or array.

Examples

```
>>> data = pd.DataFrame({"A": [1, 2, 3, 4],
...                      "B": pd.Categorical(['a', 'a', 'a', 'b'])})
>>> enc = OrdinalEncoder()
>>> trn = enc.fit_transform(data)
>>> trn
   A  B
0  1  0
1  2  0
2  3  0
3  4  1
```

```
>>> enc.columns_
Index(['A', 'B'], dtype='object')
```

```
>>> enc.non_categorical_columns_
Index(['A'], dtype='object')
```

```
>>> enc.categorical_columns_
Index(['B'], dtype='object')
```

```
>>> enc.dtypes_
{'B': CategoricalDtype(categories=['a', 'b'], ordered=False)}
```

```
>>> enc.fit_transform(dd.from_pandas(data, 2))
Dask DataFrame Structure:
                   A     B
npartitions=2
0              int64  int8
2                ...   ...
3                ...   ...
Dask Name: assign, 8 tasks
```

Methods

fit(X[, y])	Determine the categorical columns to be encoded.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Inverse ordinal-encode the columns in *X*
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Ordinal encode the categorical columns in X

__init__(*columns=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y=None*)
Determine the categorical columns to be encoded.

Parameters

X [pandas.DataFrame or dask.dataframe.DataFrame]

y [ignored]

Returns

self

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

X [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*X*)
Inverse ordinal-encode the columns in *X*

Parameters

X [array or dataframe] Either the NumPy, dask, or pandas version

Returns

data [DataFrame] Dask array or dataframe will return a Dask DataFrame. Numpy array or pandas dataframe will return a pandas DataFrame

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*X*, *y=None*)
Ordinal encode the categorical columns in X

Parameters

X [pd.DataFrame or dd.DataFrame]

y [ignored]

Returns

transformed [pd.DataFrame or dd.DataFrame] Same type as the input

dask_ml.preprocessing.LabelEncoder

class dask_ml.preprocessing.**LabelEncoder**
Encode labels with value between 0 and n_classes-1.

Read more in the User Guide.

Attributes

classes_ [array of shape (n_class,)] Holds the label for each class.

See also:

sklearn.preprocessing.OneHotEncoder encode categorical integer features using a one-hot aka one-of-K scheme.

Examples

LabelEncoder can be used to normalize labels.

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2]...)
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels.

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1]...)
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

Methods

fit(y)	Fit label encoder
fit_transform(y)	Fit label encoder and return encoded labels
get_params([deep])	Get parameters for this estimator.
inverse_transform(y)	Transform labels back to original encoding.
set_params(**params)	Set the parameters of this estimator.
transform(y)	Transform labels to normalized encoding.

__init__ (*$self, /, *args, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

fit (*y*)
Fit label encoder

Parameters

y [array-like of shape (n_samples,)] Target values.

Returns

self [returns an instance of self.]

fit_transform (*y*)
Fit label encoder and return encoded labels

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

get_params(*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*y*)
Transform labels back to original encoding.

Parameters

y [numpy array of shape [n_samples]] Target values.

Returns

y [numpy array of shape [n_samples]]

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*y*)
Transform labels to normalized encoding.

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

3.13.7 dask_ml.metrics: Metrics

Score functions, performance metrics, and pairwise distance computations.

Regression Metrics

metrics.mean_absolute_error(y_true, y_pred)	Mean squared error regression loss
metrics.mean_squared_error(y_true, y_pred[, ...])	Mean squared error regression loss
metrics.r2_score(y_true, y_pred[, ...])	R^2 (coefficient of determination) regression score function.

dask_ml.metrics.mean_absolute_error

dask_ml.metrics.**mean_absolute_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*, *compute=True*)

Mean squared error regression loss

Read more in the User Guide.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
...
array([ 0.416...,  1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.824...
```

dask_ml.metrics.mean_squared_error

dask_ml.metrics.**mean_squared_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*, *compute=True*)

Mean squared error regression loss

Read more in the User Guide.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
...
array([ 0.416...,  1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.824...
```

dask_ml.metrics.r2_score

dask_ml.metrics.**r2_score**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*, *compute=True*)

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Read more in the User Guide.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in [‘raw_values’, ‘uniform_average’, ‘variance_weighted’] or None or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is “uniform_average”.

‘raw_values’ : Returns a full set of scores in case of multioutput input.

‘uniform_average’ : Scores of all outputs are averaged with uniform weight.

‘variance_weighted’ : Scores of all outputs are averaged, weighted by the variances of each individual output.

Changed in version 0.19: Default value of multioutput is ‘uniform_average’.

Returns

z [float or ndarray of floats] The R^2 score or ndarray of scores if ‘multioutput’ is ‘raw_values’.

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

References

[1]

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred, multioutput='variance_weighted')
...
0.938...
>>> y_true = [1,2,3]
>>> y_pred = [1,2,3]
>>> r2_score(y_true, y_pred)
1.0
>>> y_true = [1,2,3]
>>> y_pred = [2,2,2]
>>> r2_score(y_true, y_pred)
0.0
>>> y_true = [1,2,3]
>>> y_pred = [3,2,1]
>>> r2_score(y_true, y_pred)
-3.0
```

Classification Metrics

metrics.accuracy_score(y_true, y_pred[, ...])	Accuracy classification score.

dask_ml.metrics.accuracy_score

dask_ml.metrics.**accuracy_score**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*, *compute=True*)

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the User Guide.

Parameters

y_true [1d array-like, or label indicator array] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.

sample_weight [1d array-like, optional] Sample weights.

New in version 0.7.0.

Returns

score [scalar dask Array] If `normalize == True`, return the correctly classified samples (float), else it returns the number of correctly classified samples (int).

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

Notes

In binary and multiclass classification, this function is equal to the `jaccard_similarity_score` function.

Examples

```
>>> import dask.array as da
>>> import numpy as np
>>> from dask_ml.metrics import accuracy_score
>>> y_pred = da.from_array(np.array([0, 2, 1, 3]), chunks=2)
>>> y_true = da.from_array(np.array([0, 1, 2, 3]), chunks=2)
>>> accuracy_score(y_true, y_pred)
dask.array<mean_agg-aggregate, shape=(), dtype=float64, chunksize=()>
>>> _.compute()
0.5
>>> accuracy_score(y_true, y_pred, normalize=False).compute()
2
```

In the multilabel case with binary label indicators:

```
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

3.13.8 dask_ml.tensorflow: Tensorflow

start_tensorflow

3.13.9 dask_ml.xgboost: XGBoost

Train an XGBoost model on dask arrays or dataframes.

This may be used for training an XGBoost model on a cluster. XGBoost will be setup in distributed mode alongside your existing `dask.distributed` cluster.

XGBClassifier([max_depth, learning_rate, ...])	**Attributes**
XGBRegressor([max_depth, learning_rate, ...])	**Attributes**

dask_ml.xgboost.XGBClassifier

class dask_ml.xgboost.**XGBClassifier**(*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='binary:logistic'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, ***kwargs*)

Attributes

feature_importances_ Returns

Methods

apply(X[, ntree_limit])	Return the predicted leaf every tree for each sample.
evals_result()	Return the evaluation results.
fit(X[, y])	Fit a gradient boosting classifier
get_booster()	Get the underlying xgboost Booster of this model.
get_params([deep])	Get parameters.
get_xgb_params()	Get xgboost type parameters.
predict(X)	Predict with *data*.
predict_proba(data[, ntree_limit])	Predict the probability of each *data* example being of a given class.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

__init__ (*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='binary:logistic'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

apply (*X*, *ntree_limit=0*)
Return the predicted leaf every tree for each sample.

Parameters

X [array_like, shape=[n_samples, n_features]] Input features matrix.

ntree_limit [int] Limit number of trees in the prediction; defaults to 0 (use all trees).

Returns

X_leaves [array_like, shape=[n_samples, n_trees]] For each datapoint x in X and for each tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

evals_result ()
Return the evaluation results.

If eval_set is passed to the *fit* function, you can call evals_result() to get evaluation results for all passed eval_sets. When eval_metric is also passed to the *fit* function, the evals_result will contain the eval_metrics passed to the *fit* function

Returns

evals_result [dictionary]

feature_importances_

Returns

feature_importances_ [array of shape = [n_features]]

fit (*X*, *y=None*)
Fit a gradient boosting classifier

Parameters

X [array-like [n_samples, n_features]] Feature Matrix. May be a dask.array or dask.dataframe

y [array-like] Labels

Returns

self [XGBClassifier]

Notes

This differs from the XGBoost version in three ways

1. The `sample_weight`, `eval_set`, `eval_metric`, `early_stopping_rounds` and `verbose` fit kwargs are not supported.
2. The labels are not automatically label-encoded
3. The `classes_` and `n_classes_` attributes are not learned

get_booster()
Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns

booster [a xgboost booster of underlying model]

get_params(*deep=False*)
Get parameters.

get_xgb_params()
Get xgboost type parameters.

predict(*X*)
Predict with *data*. NOTE: This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict

data [DMatrix] The dmatrix storing the input.

output_margin [bool] Whether to output the raw untransformed margin value.

ntree_limit [int] Limit number of trees in the prediction; defaults to 0 (use all trees).

prediction : numpy array

predict_proba(*data*, *ntree_limit=0*)
Predict the probability of each *data* example being of a given class. NOTE: This function is not thread safe.

For each booster object, predict can only be called from one thread. If you want to run prediction using multiple thread, call xgb.copy() to make copies of model object and then call predict

data [DMatrix] The dmatrix storing the input.

ntree_limit [int] Limit number of trees in the prediction; defaults to 0 (use all trees).

prediction [numpy array] a numpy array with the probability of each data example being of a given class.

score(*X*, *y*, *sample_weight=None*)
Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

dask_ml.xgboost.XGBRegressor

class dask_ml.xgboost.**XGBRegressor**(*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='reg:linear'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, ***kwargs*)

Attributes

feature_importances_ Returns

Methods

apply(X[, ntree_limit])	Return the predicted leaf every tree for each sample.
evals_result()	Return the evaluation results.
fit(X[, y])	Fit the gradient boosting model
get_booster()	Get the underlying xgboost Booster of this model.
get_params([deep])	Get parameters.
get_xgb_params()	Get xgboost type parameters.
score(X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

predict	

__init__(*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='reg:linear'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

apply(*X*, *ntree_limit=0*)

Return the predicted leaf every tree for each sample.

Parameters

X [array_like, shape=[n_samples, n_features]] Input features matrix.

ntree_limit [int] Limit number of trees in the prediction; defaults to 0 (use all trees).

Returns

X_leaves [array_like, shape=[n_samples, n_trees]] For each datapoint x in X and for each

tree, return the index of the leaf x ends up in. Leaves are numbered within `[0; 2**(self.max_depth+1))`, possibly with gaps in the numbering.

evals_result()
Return the evaluation results.

If eval_set is passed to the *fit* function, you can call evals_result() to get evaluation results for all passed eval_sets. When eval_metric is also passed to the *fit* function, the evals_result will contain the eval_metrics passed to the *fit* function

Returns

evals_result [dictionary]

feature_importances_

Returns

feature_importances_ [array of shape = [n_features]]

fit(*X*, *y=None*)
Fit the gradient boosting model

Parameters

X [array-like [n_samples, n_features]]

y [array-like]

Returns

self [the fitted Regressor]

Notes

This differs from the XGBoost version not supporting the `eval_set`, `eval_metric`, `early_stopping_rounds` and `verbose` fit kwargs.

get_booster()
Get the underlying xgboost Booster of this model.

This will raise an exception when fit was not called

Returns

booster [a xgboost booster of underlying model]

get_params(*deep=False*)
Get parameters.

get_xgb_params()
Get xgboost type parameters.

score(*X*, *y*, *sample_weight=None*)
Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

train(client, params, data, labels[, ...])	Train an XGBoost model on a Dask Cluster
predict(client, model, data)	Distributed prediction with XGBoost

dask_ml.xgboost.train

dask_ml.xgboost.**train**(*client*, *params*, *data*, *labels*, *dmatrix_kwargs={}*, ***kwargs*)
Train an XGBoost model on a Dask Cluster

This starts XGBoost on all Dask workers, moves input data to those workers, and then calls `xgboost.train` on the inputs.

Parameters

client: dask.distributed.Client

params: dict Parameters to give to XGBoost (see xgb.Booster.train)

data: dask array or dask.dataframe

labels: dask.array or dask.dataframe

dmatrix_kwargs: Keywords to give to Xgboost DMatrix

****kwargs: Keywords to give to XGBoost train**

See also:

predict

Examples

```
>>> client = Client('scheduler-address:8786')
>>> data = dd.read_csv('s3://...')
>>> labels = data['outcome']
>>> del data['outcome']
>>> train(client, params, data, labels, **normal_kwargs)
<xgboost.core.Booster object at ...>
```

dask_ml.xgboost.predict

dask_ml.xgboost.**predict**(*client*, *model*, *data*)

Distributed prediction with XGBoost

Parameters

client: dask.distributed.Client

model: xgboost.Booster

data: dask array or dataframe

Returns

Dask.dataframe or dask.array, depending on the input data type

See also:

train

Examples

```
>>> client = Client('scheduler-address:8786')
>>> test_data = dd.read_csv('s3://...')
>>> model
<xgboost.core.Booster object at ...>
```

```
>>> predictions = predict(client, model, test_data)
```

3.14 Changelog

3.14.1 Version 0.8.0

Enhancements

- Automatically replace default scikit-learn scorers with dask-aware versions in Incremental (GH#200)

3.14.2 Version 0.7.0

Enhancements

- Added `sample_weight` support for *dask_ml.metrics.accuracy_score()*. (GH#217)
- Improved performance of training on *dask_ml.cluster.SpectralClustering* (GH#152)
- Added *dask_ml.preprocessing.LabelEncoder*. (GH#226)
- Fixed issue in `model_selection` meta-estimators not respecting the default Dask scheduler (GH#260)

API Breaking Changes

- Removed the `basis_inds_` attribute from *dask_ml.cluster.SpectralClustering* as its no longer used (GH#152)
- Change *dask_ml.wrappers.Incremental.fit()* to clone the underlying estimator before training (GH#258). This induces a few changes
  1. The underlying estimator no longer gives access to learned attributes like `coef_`. We recommend using `Incremental.coef_`.
  2. State no longer leaks between successive `fit` calls. Note that `Incremental.partial_fit()` is still available if you want state, like learned attributes or random seeds, to be re-used. This is useful if you're making multiple passes over the training data.
- Changed `get_params` and `set_params` for *dask_ml.wrappers.Incremental* to no longer magically get / set parameters for the underlying estimator (GH#258). To specify parameters for the underlying estimator, use the double-underscore prefix convention established by scikit-learn:

```
inc.set_params('estimator__alpha': 10)
```

Reorganization

Dask-SearchCV is now being developed in the `dask/dask-ml` repository. Users who previously installed `dask-searchcv` should now just install `dask-ml`.

Bug Fixes

- Fixed random seed generation on 32-bit platforms (GH#230)

3.14.3 Version 0.6.0

API Breaking Changes

- Removed the *get* keyword from the incremental learner `fit` methods. (GH#187)
- Deprecated the various `Partial*` estimators in favor of the *dask_ml.wrappers.Incremental* meta-estimator (GH#190)

Enhancements

- Added a new meta-estimator *dask_ml.wrappers.Incremental* for wrapping any estimator with a *partial_fit* method. See *Incremental Meta-estimator* for more. (GH#190)
- Added an R2-score metric *dask_ml.metrics.r2_score()*.

3.14.4 Version 0.5.0

API Breaking Changes

- The *n_samples_seen_* attribute on `dask_ml.preprocessing.StandardScalar` is now consistently `numpy.nan` (GH#157).

- Changed the algorithm for `dask_ml.datasets.make_blobs()`, `dask_ml.datasets.make_regression()` and `dask_ml.datasets.make_classfication()` to reduce the single-machine peak memory usage (GH#67)

Enhancements

- Added *dask_ml.model_selection.train_test_split()* and *dask_ml.model_selection.ShuffleSplit* (GH#172)
- Added `dask_ml.metrics.classification_score()`, *dask_ml.metrics.mean_absolute_error()*, and *dask_ml.metrics.mean_squared_error()*.

Bug Fixes

- `dask_ml.preprocessing.StandardScalar` now works on DataFrame inputs (GH#157).
-

3.14.5 Version 0.4.1

This release added several new estimators.

Enhancements

Added `dask_ml.preprocessing.RobustScaler`

Scale features using statistics that are robust to outliers. This mirrors `sklearn.preprocessing.RobustScalar` (GH#62).

Added `dask_ml.preprocessing.OrdinalEncoder`

Encodes categorical features as ordinal, in one ordered feature (GH#119).

Added `dask_ml.wrappers.ParallelPostFit`

A meta-estimator for fitting with any scikit-learn estimator, but post-processing (`predict`, `transform`, etc.) in parallel on dask arrays. See *Parallel Meta-estimators* for more (GH#132).

3.14.6 Version 0.4.0

API Changes

- Changed the arguments of the dask-glm based estimators in `dask_glm.linear_model` to match scikit-learn's API (GH#94).
  - To specify `lambuh` use `C = 1.0 / lambduh` (the default of 1.0 is unchanged)
  - The `rho`, `over_relax`, `abstol` and `reltol` arguments have been removed. Provide them in `solver_kwargs` instead.

This affects the `LinearRegression`, `LogisticRegression` and `PoissonRegression` estimators.

Enhancements

- Accept `dask.dataframe` for dask-glm based estimators (GH#84).

3.14.7 Version 0.3.2

Enhancements

- Added `dask_ml.preprocessing.TruncatedSVD()` and `dask_ml.preprocessing.PCA()` (GH#78)

3.14.8 Version 0.3.0

Enhancements

- Added `KMeans.predict()` (GH#83)

API Changes

- Changed the fitted attributes on `MinMaxScaler` and `StandardScaler` to be concrete NumPy or pandas objects, rather than persisted dask objects (GH#75).

3.15 Contributing

Thanks for helping to build `dask-ml`!

3.15.1 Cloning the Repository

Make a fork of the dask-ml repo and clone the fork

```
git clone https://github.com/<your-github-username>/dask-ml
cd dask-ml
```

You may want to add `https://github.com/dask/dask-ml` as an upstream remote repository.

```
git remote add upstream https://github.com/dask/dask-ml
```

3.15.2 Creating an environment

We have conda environment YAML files with all the necessary dependencies in the `ci` directory. If you're using conda you can

```
conda env create -f ci/environment-3.6.yml --name=dask-ml-dev
```

to create a conda environment and install all the dependencies. Note there is also a `ci/environment-2.7.yml` file if you need to use Python 2.7.

If you're using pip, you can view the list of all the required and optional dependencies within `setup.py` (see the `install_requires` field for required dependencies and `extras_require` for optional dependencies). You'll at least need the build dependencies of NumPy, setuptools, setuptools_scm, and Cython.

3.15.3 Building dask-ml

The library has some C-extensions, so installing is a bit more complicated than normal. If you have a compiler and everything is setup correctly, you should be able to install Cython and all the required dependencies.

From within the repository:

```
python setup.py build_ext --inplace
```

And then

```
python -m pip install -e ".[dev]"
```

If you have any trouble with the build step, please open an issue in the dask-ml issue tracker.

3.15.4 Running tests

Dask-ml uses py.test for testing. You can run tests from the main dask-ml directory as follows:

```
py.test tests
```

Alternatively you may choose to run only a subset of the full test suite. For example to test only the preprocessing submodule we would run tests as follows:

```
py.test tests/preprocessing
```

In addition to running tests, dask-ml verifies code style uniformity with the `flake8` tool:

```
pip install flake8
flake8
```

3.15.5 Conventions

For the most part, we follow scikit-learn's API design. If you're implementing a new estimator, it will ideally pass scikit-learn's estimator check.

We have some additional decisions to make in the dask context. Ideally

1. All attributes learned during `.fit` should be *concrete*, i.e. they should not be dask collections.
2. To the extent possible, transformers should support
    - `numpy.ndarray`
    - `pandas.DataFrame`
    - `dask.Array`
    - `dask.DataFrame`

3. If possible, transformers should accept a `columns` keyword to limit the transformation to just those columns, while passing through other columns untouched. `inverse_transform` should behave similarly (ignoring other columns) so that `inverse_transform(transform(X))` equals `X`.
4. Methods returning arrays (like `.transform`, `.predict`), should return the same type as the input. So if a `dask.array` is passed in, a `dask.array` with the same chunks should be returned.

3.16 History

Dask in machine learning originally consisted of a number of smaller libraries focused around particular sub-domains of machine learning.

- dask-searchcv: Scalable model selection
- dask-glm: Generalized Linear Model solvers
- dask-xgboost: Connection to the XGBoost library
- dask-tensorflow: Connection to the Tensorflow library

While these special-purpose libraries were convenient for development, they were inconvenient for users who found the number of libraries daunting. The `dask-ml` project started as a combination of these that presented a single unified API and entry-point that mimicked Scikit-Learn. Afterwards additional algorithm development happened in the `dask-ml` library itself.

The pre-existing libraries are still valid and dask-ml defers to them for future development.

Bibliography

[1] Wikipedia entry on the Coefficient of determination

Python Module Index

d

Index

Symbols

A

C

D

E

F

G

T

V

X